

# Applied Choreographies

Maurizio Gabbrielli    Saverio Giallorenzo

University of Bologna — Department of Computer  
Science and Engineering

gabbri@cs.unibo.it    sgiallor@cs.unibo.it

Fabrizio Montesi

University of Southern Denmark — Department of  
Mathematics and Computer Science

fmontesi@gmail.com

## Abstract

Choreographic Programming is a methodology for the development of concurrent software based on a correctness-by-construction approach which, given a global description of a system (a choreography), automatically generates deadlock-free communicating programs via an EndPoint Projection (EPP). Previous works use target-languages for EPP that, like their source choreography languages, model communications using channel names (e.g., variants of CCS and  $\pi$ -calculus). This leaves a gap between such models and real-world implementations, where communications are concretely supported by low-level mechanisms for message routing.

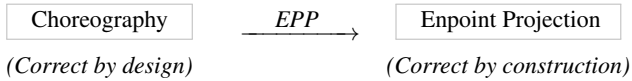
We bridge this gap by developing Applied Choreographies (AC), a new model for choreographic programming. AC brings the correctness-by-construction methodology of choreographies down to the level of a real executable language. The key feature of AC is that its semantics is based on message correlation — a standard technique in Service-Oriented Computing — while retaining the usual simple and intuitive syntax of choreography languages. We provide AC with a typing discipline that ensures the correct use of the low-level mechanism of message correlation, thus avoiding communication errors. We also define a two-step compilation from AC to a low-level Correlation Calculus, which is the basis of a real executable language (Jolie). Finally, we prove an operational correspondence theorem, which ensures that compiled programs behave as the original choreography. This is the first result of such correctness property in the case of a real-world implemented language.

**Categories and Subject Descriptors** D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.2 [Semantics of Programming Languages]: Process Models

**Keywords** Concurrency, Types, Session, Correlation

## 1. Introduction

Choreographic Programming is a methodology for the development of concurrent software, based on a correctness-by-construction approach that we can depict as follows:



Above, a choreography language allows to write programs, called *choreographies*, which are global, high-level abstract descriptions of the communications enacted by the endpoint processes in a sys-

tem [25]. Executable programs are then automatically obtained from choreographies by means of *EndPoint Projection* (EPP) [8–10, 22, 34]. EPP transforms the global descriptions into endpoint programs with local I/O actions for message passing. Correctness by construction then follows from the fact that the syntax of a choreography language does not allow to write mismatched I/O actions and that EPP preserves this property, thus ensuring that the generated endpoint code is deadlock-free. Several works used choreographies for standards [1, 36], language implementations [2, 12, 18, 31, 35], and to ease the automatic detection of programming errors [3, 9, 10, 22, 33]. Recent results showed that choreographic programming can be used to deal with important practical aspects of distributed programming, such as asynchrony, parallelism, modularity [8, 27], and adaptation [33].

A key point in standard proofs of correctness of EPP is to use as target language a process calculus close to the source choreography language [8–10, 17, 22, 34]. In particular, these endpoint calculi model communications through synchronisations on *names* (as in CCS and the  $\pi$ -calculus [23, 24]), abstracting from how real-world frameworks actually support communications. Consequently, implementations of choreographic programming significantly depart from such formal models, as they have to deal with the design of key mechanisms such as message routing and the creation of new channels (as typically happens in the implementation of process calculi, see, e.g., [11, 19]). As an example, consider the Chor and AIOJ languages [2, 12]: they both implement a formal model based on synchronisation on names (respectively [8] and [33]) but their implementation of EPP targets Jolie [21, 28], a Service-Oriented language that defines communication behaviour on *message correlation*<sup>1</sup>. This makes the EPP implementations of Chor and AIOJ much more technically involved than their formal specifications, including the management of underlying data structures (e.g., message queues) and unexpected additional communications in the resulting executable code. This key difference between formal models and implementations can compromise the benefits of choreographic programming: the correctness-by-construction approach and the clear specification of the communications carried out during execution. Thus we ask:

*How can we formalise the implementation of communications in choreography languages?*

Clearly, a satisfactory answer should preserve the correctness-by-construction guarantees of choreography models down to the level of how communications are concretely implemented. A challenging task that requires the definition of a model with the typical clarity of choreography languages, also providing all the necessary details to formally reason about how communications are supported at the lower level in a tractable way.

Our answer is to develop a theory of *Applied Choreographies* (AC), based on notions from the setting of Service-Oriented Computing: the setting where choreographies are used the most as design tool [1, 36]. The key contribution of AC lies in its semantics

[Copyright notice will appear here once 'preprint' option is removed.]

<sup>1</sup> A standard technique in Service-Oriented Computing and Web Services that routes messages inspecting the data they carry (e.g., headers) [26, 30].

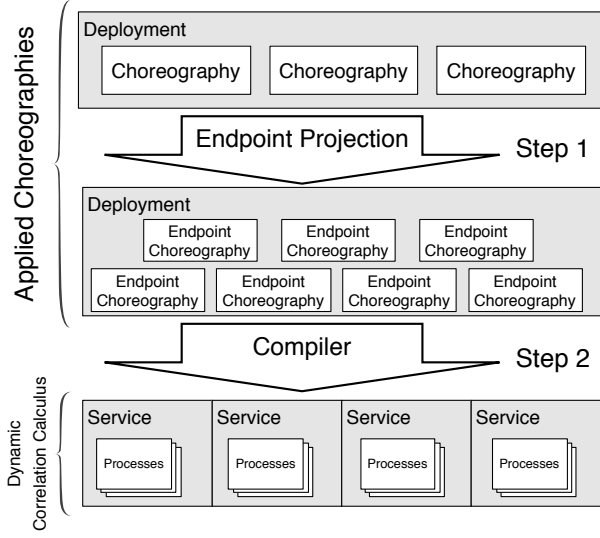


Figure 1: Overall methodology of Applied Choreographies.

that, equipped with a novel notion of deployment, allows us to reason about how messages are routed from a participant to another through message correlation. On this result, we could define a correct executable language, Jolie. Nevertheless, AC provides the usual simple and intuitive syntax of choreography languages and also the most advanced features of recent choreography models, such as modularity, asynchrony, parallelism, and the dynamic creation of multiparty sessions and processes.

**Contributions.** We list below our main contributions:

**Applied Choreographies.** We introduce Applied Choreographies, a choreography model for the modular development of choreographies based on asynchronous message passing. AC captures how communications among processes are concretely supported via message correlation (§ 2). The main novel aspect of AC is a notion of *deployment* for choreographies which, for each process in the choreography, keeps track of *i*) the location of the service in which it executes (multiple processes can run in the same service), *ii*) its input message queues (for asynchronous communications), and *iii*) its state (the value of its variables). This yields a close representation of how real-world service-oriented scenarios implement communications, which is the basis for all of our results.

**Type system.** Since the semantics of AC models how messages are passed from a sender to a receiver, AC programs may encounter execution errors. For example, it may be impossible to route a message from one endpoint to another, because of missing information in the deployment of a choreography (e.g., a suitable input queue at the receiver). This is a significant departure from previous choreography models, where it is assumed that communications between two processes can always be performed [3, 8, 9, 17, 22, 33, 34]. In § 3, we present a typing discipline for AC that prevents communication errors. Interestingly, the additional complexity of the semantics of AC does not increase the complexity of the types that the programmer has to specify to verify a choreography: our type language is that of standard multiparty session types [17].

**Compilation of AC programs.** We define a two-step methodology to transform a choreography in AC into programs of the *Dynamic Correlation Calculus* (DCC), a model for executable code based on message correlation. We depict our methodology in Figure 1. The first step is a source-to-source Endpoint Projection (EPP) (as seen in [27]) which projects a choreography describing the behaviour of many participants to a series of choreography modules, called *Endpoint Choreographies*, each describing the behaviour of a single participant. The second step is a compilation from endpoint

choreographies to DCC programs, which define the behaviour of *services* following the Service-Oriented model. Specifically, DCC formalises a syntax and a semantics for a fragment of the Jolie language. We chose Jolie because its reference implementation equips a formal specification [26] called Correlation Calculus (CC). DCC improves CC by adding message queues that can be created at run-time, a necessary feature to support our choreography model. We prove that the compiled processes implement the behaviour of the EPP and therefore, that of the originating AC program. The supplemental material contains full definitions and examples.

## 2. Applied Choreography Language

This section introduces Applied Choreographies (AC). On key elements of other choreography calculi like *processes*, *sessions*, and *roles*, AC introduces the notion of *location*, as described below.

### 2.1 Syntax

In the syntax of AC (Figure 2)  $C$  denotes a choreography,  $p, q$  processes,  $A, B$  roles,  $k$  sessions,  $o$  operations,  $l$  locations,  $X$  procedures, and  $x$  variables<sup>2</sup>. We consider all sets of identifiers disjoint. Processes are independent execution units that proceed in parallel. They can communicate with each other through sessions. Roles track which role each process plays in a session. As in standard multiparty session types [17] Roles are the basis for our typing discipline in § 3. Locations represent publicly reachable addresses, where we assume that an always-available service supports the creation and the execution of new processes. We introduce the notion of location to model the deployment of processes into services. Following the SOC model, a service is a container, reachable at a defined address (the location), where processes execute in parallel. In § 6, when compiling ACs to the lower language DCC, we map each location to a concrete service implementation. Using locations makes us depart from previous choreography languages, which do not consider the deployment of processes in their models.

AC includes complete and partial actions to support modularity, as in previous choreography models [27]. A complete action specifies the behaviour of all participants involved in the action. A partial action describes the behaviour of only some participants. Partial actions enable compositionality: choreographies with compatible partial actions can be composed in parallel.

**Complete Actions.** Term (*start*) denotes session initiation: process  $p$  starts a new multiparty session  $k$  together with processes  $\tilde{q}$ . Process  $p$ , called *active process*, is already running, whereas each process  $q$  in  $\tilde{l}.q$ , called *service process*, is dynamically created at its respective location  $l$  in  $\tilde{l}.q$ . Service locations can be used repeatedly to spawn multiple processes, even inside of recursions. We assume  $\tilde{l}.q$  always non-empty. Term (*com*) models a communication: on session  $k$ , process  $p$  sends to process  $q$  a message for operation  $o$ , carrying the evaluation of expression  $e$  in the local state of  $p$ ; process  $q$  stores the received value in its local variable  $x$ .

**Partial Actions.** In term (*req*), process  $p$  requests the creation of some external processes at their respective locations  $\tilde{l}$  to start a new session  $k$ .  $\tilde{l}.B$  means that each location  $l$  in  $\tilde{l}.B$  is expected to spawn a process that behaves as specified by the related role  $B$ . The dual of (*req*) is term (*acc*), which provides the implementation of service processes. Specifically, term (*acc*) defines a reusable choreography module that accepts, at locations  $\tilde{l}$ , the creation of processes  $\tilde{q}$  playing their respective roles  $\tilde{B}$ . Following the design idea that services should always be available, shared by other models [8, 9], we assume that all (*acc*) terms in a choreography are at top level (not guarded by other actions). Term (*send*) models the sending of a message, for operation  $o$ , from process  $p$  to an external process playing role  $B$  in session  $k$ . Dually, in term (*recv*), process  $q$

<sup>2</sup> Variables are *paths* to traverse structured data, e.g.,  $x$  can be a path  $\mathbf{x.y.z}$  where “.” is the path nesting operator. § 2.2 formalises variables and paths.

$C ::= \eta; C$	$(seq) \mid C_1 \mid C_2$	$(par)$
$\mid k : A \rightarrow q[B].\{o_i(x_i); C_i\}_{i \in I}$	$(recv) \mid \text{def } X(\bar{p}) = C' \text{ in } C$	$(def)$
$\mid \text{acc } k : l.q[B]; C$	$(acc) \mid X(\bar{p})$	$(call)$
$\mid \text{if } p.e \{C_1\} \text{ else } \{C_2\}$	$(cond) \mid \mathbf{0}$	$(inact)$
$\eta ::= \text{start } k : p[A] \Leftrightarrow l.q[B]$	$(start) \mid \text{req } k : p[A] \Leftrightarrow l.B$	$(req)$
$\mid k : p[A].e \rightarrow q[B].o(x)$	$(com) \mid k : p[A].e \rightarrow B.o$	$(send)$

Figure 2: Choreography Calculus - Syntax

receives a message from an external process playing role A in session  $k$ ; depending on the operation  $o_i$  that the message is for,  $q$  proceeds with the respective continuation  $C_i$  (this is the standard branching found in session-oriented calculi [16]).

**Other Terms.** In a conditional (*cond*) process  $p$  evaluates a condition  $e$  in its local state to choose between the continuations  $C_1$  and  $C_2$ . Term (*par*) is the standard parallel composition of choreographies, as in [10, 27]. Terms (*def*), (*call*), and (*inact*) denote respectively the standard definition of recursive procedures, procedure calls, and inaction. Some terms bind identifiers in continuations. In terms (*start*) and (*acc*), the session identifier  $k$  and the process identifiers  $\bar{q}$  are bound (as they are freshly created). All other identifiers are free. In terms (*com*) and (*recv*), the variables used by the receiver to store the message are bound ( $x$  and all the  $x_i$ , respectively). In term (*req*), the session identifier  $k$  is bound. Finally, in term (*def*), the procedure identifier  $X$  is bound. In the remainder, we sometimes omit  $\mathbf{0}$  or irrelevant variables (e.g., in communications with empty messages).

**REMARK 1.** As in [8], except for terms (*start*), (*req*), and (*acc*), annotating roles in AC is not technically necessary since we can infer roles from session identifiers. However, all AC terms include roles to simplify the presentation of our typing discipline in § 3.

## 2.2 Semantics

The semantics of AC is one of our major contributions: it formally captures, on the level of choreographies, the real-world communication mechanism found in SOC, called message correlation [30]. We first give an informal overview of this communication mechanism. Communications in SOC are asynchronous: each process has a set of FIFO input queues that act as buffers, managed by its enclosing service. Each queue is equipped with some data, here called *correlation key*, that can be used to distinguish (identify) the queue among the many present inside of a service. When a service receives a message from the network, it inspects the content of the message for a portion of data that matches the correlation key of one of its queues. If a queue can be found, the message is inserted at the end of it. The process owning the queue will be able to consume the message later on in its execution. Thus, when a sender process  $p$  sends a message to a receiver process  $q$ , it needs to know *i*) the location of the service where  $q$  is running and *ii*) the correlation key of one of the queues owned by  $q$ . In practice, the correlation key in the message can be a part of the message payload itself or be in some separate headers; in this work, we abstract from this detail.

Below, we formalise the semantics of AC by equipping choreographies with *deployments*, ranged over by  $D$ . We use deployments to formalise the elements of SOC that we have just informally described, in particular the *state* and *message queues* of processes located at services. We define each element separately.

**Data and Process state.** Data in SOC is typically structured following a tree-like format, e.g., XML [6] and JSON [5]. In this work, we use trees to represent both messages and the state of running processes (as in [26]). Formally, we consider a set  $\mathcal{T}$  of rooted trees, ranged over by  $t$ , where edges are labelled by names, ranged over by  $\underline{x}, \underline{y}, \dots$ . We assume that all outgoing edges of a node have distinct labels and that only leafs contain values, which can be either a location  $l$  or some basic data (integer, string, etc.). Variables in AC,

ranged over by  $x$ , are formalised as paths to traverse a tree:

$$x, y, z ::= \underline{x}.x \mid \varepsilon$$

$\varepsilon$  is the empty path; we often omit the trailing  $\varepsilon$  in paths. Given a path  $x$  and a nonempty tree  $t$ , we denote by  $x(t)$  the node reached following the path  $x$  in  $t$ . Observe that  $x(t)$  is partially defined since the path  $x$  may not be valid in  $t$  in general. By a slight abuse of notation, when  $x(t)$  is a leaf we denote by  $x(t)$  also the value of the node. In our semantics, we will also use the replacement operator  $t \triangleleft (x, t')$ . If  $x(t)$  is defined,  $t \triangleleft (x, t')$  returns the tree obtained replacing in  $t$  the subtree rooted in  $x(t)$  by  $t'$ . If  $x(t)$  is undefined,  $t \triangleleft (x, t')$  adds the smallest chain of empty nodes to  $t$  such that  $x(t)$  is defined and insert  $t'$ .

**Deployment.** A deployment  $D$  is an overloaded partial function on locations and processes. Intuitively, locations are mapped to the set of processes running at that location, whereas processes are mapped to their local state and input queues. Therefore, given a location  $l$ , we read  $D(l) = \{\bar{p}\}$  as “the processes  $\bar{p}$  are running at the location  $l$ ” (for any process  $p$  and deployment  $D$ , we assume that each process  $p$  is always located at only one location). For processes, instead, given a process  $p$  then  $D(p)$  returns a pair  $(t, M)$ , where  $t$  is a tree representing the local state of  $p$  and  $M$  is a *queue map*. A queue map defines the input queues that  $p$  can use to receive messages from other processes. Formally, a queue map  $M$  is a partial function of type  $M : \mathcal{T} \rightarrow \text{Seq}(O \times \mathcal{T})^3$ .

A map  $M(t_c) = (\bar{o}, t)$  means that the process owning  $M$  has an input queue containing  $(\bar{o}, t)$  (an ordered sequence of messages<sup>4</sup>) that *correlates* with the data  $t_c$ , called the correlation key of the queue. In our semantics, we will use correlation keys to formalise our mechanism of message correlation: when an incoming message with some correlation data arrives from a sender at a location, our semantics will place the message in a queue that has the same correlation data of the message as correlation key. In a message  $(o, t)$ ,  $o$  is the operation used by the sender and  $t$  is the payload of the message. In the remainder of the paper, for  $D(p) = (t, M)$ , we use the shortcuts  $D(p).st$  to refer to  $t$  (the state of  $p$ ) and  $D(p).que$  to refer to  $M$  (the queues of  $p$ ), respectively.

Observe that, in our model, programmers do not need to specify the deployment of a choreography<sup>5</sup>. Concretely, for any choreography program  $C$  with no free session names (all sessions are under a start term), we can define a *default deployment* where all active processes are assigned to some location and given an empty state and queue map. Below, we denote the set of free process names in a choreography  $C$  with  $\text{fp}(C)$ , and we write  $t_\perp$  for the empty tree.

**DEFINITION 1 (Default Deployment).** Let  $C$  be a choreography with no free session names. Then,  $D$  is a default deployment for  $C$  if for all  $p \in \text{fp}(C) : D(p) = (t_\perp, \emptyset)$  and  $p \in D(l)$  for some  $l$ .

**Reductions.** We present the semantics for AC in terms of reductions of the form  $D, C \rightarrow D', C'$ , where  $D, C$  is a *running choreography*. Formally, the reduction relation  $\rightarrow$  is the smallest closed under the rules reported in Figure 3. In the rules for session creation and communications, we make use of the auxiliary relation  $D, \delta \blacktriangleright D'$  to model the effect of an action  $\delta$  on a deployment  $D$ , resulting in a new deployment  $D'$ .  $\delta$  is defined as:

$$\delta ::= \text{start } k : l.p[A] \mid k : p[A].e \rightarrow B.o \mid k : A \rightarrow q[B].o(\underline{x})$$

denoting, from left to right, the start of a session and sending and reception of a message. Below we separately discuss our main rules for  $D, C \rightarrow D', C'$ , along the definition of  $D, \delta \blacktriangleright D'$  for each  $\delta$ .

**Rule  $[^C]_{\text{START}}$ .** Rule  $[^C]_{\text{START}}$  starts a fresh (denoted by  $\#$ ) session  $k'$  with a complete action, together with some fresh processes

<sup>3</sup>  $\text{Seq}(-)$  is the type constructor of sequences.

<sup>4</sup> We denote the empty list  $\varepsilon$  and a list of messages  $(o_1, t_1) :: \dots :: (o_n, t_n)$ . We use  $::$  rather than commas for a clearer definition of the semantics of AC.

<sup>5</sup> As in other languages states of programs are not specified by programmers.

$$\begin{array}{c}
\frac{\eta = k : p[A].e \rightarrow B.o \quad D, \eta \triangleright D'}{D, \eta; C \rightarrow D', C} [C]_{\text{SEND}} \quad \frac{j \in I \quad D, k : A \rightarrow q[B].o_j(x_j) \triangleright D'}{D, k : A \rightarrow q[B].\{o_i(x_i); C_i\}_{i \in I} \rightarrow D', C_j} [C]_{\text{RECV}} \\
\frac{\eta = k : p[A].e \rightarrow q[B].o(x) \quad D, k : p[A].e \rightarrow B.o \triangleright D'}{D, \eta; C \rightarrow D', k : A \rightarrow q[B].o(x); C} [C]_{\text{COM}} \quad \frac{i = 1 \text{ if } \text{eval}(e, D(p).\text{st}) = \text{true}, i = 2 \text{ otherwise}}{D, \text{if } p.e \{C_1\} \text{ else } \{C_2\} \rightarrow D, C_i} [C]_{\text{COND}} \\
\frac{D, C_1 \rightarrow D', C'_1}{D, \text{def } X = C_2 \text{ in } C_1 \rightarrow D', \text{def } X = C_2 \text{ in } C'_1} [C]_{\text{CTX}} \quad \frac{\mathcal{R} \in \{\equiv, \simeq\} \quad C_1 \mathcal{R} C'_1 \quad D, C'_1 \rightarrow D', C'_2 \quad C'_2 \mathcal{R} C_2}{D, C_1 \rightarrow D', C_2} [C]_{\text{EQ}} \\
\frac{D, C_1 \rightarrow D', C'_1}{D, C_1 | C_2 \rightarrow D', C'_1 | C_2} [C]_{\text{PAR}} \quad \frac{\# \tilde{r} \quad \# k' \quad p \in D(l) \quad \delta = \text{start } k' : l.p[A], \tilde{l}.r[B] \quad D, \delta \triangleright D'}{D, \text{start } k : p[A] \Leftrightarrow \tilde{l}.q[B]; C \rightarrow D', C[k'/k][\tilde{r}/\tilde{q}]} [C]_{\text{START}} \\
\frac{i \in \{1, \dots, n\} \quad \# k' \quad \{\tilde{l}.B\} = \bigcup_i \{\tilde{l}.B_i\} \quad \# \tilde{r} \quad \{\tilde{r}\} = \bigcup_i \{\tilde{r}_i\} \quad p \in D(l) \quad \delta = \text{start } k' : l.p[A], \tilde{l}.r_1[B_1], \dots, \tilde{l}.r_n[B_n] \quad D, \delta \triangleright D'}{D, \text{req } k : p[A] \Leftrightarrow \tilde{l}.B; C | \prod_i (\text{acc } k : \tilde{l}.q_i[B_i]; C_i) \rightarrow D', C[k'/k] | \prod_i (C_i[k'/k][\tilde{r}_i/\tilde{q}_i]) | \prod_i (\text{acc } k : \tilde{l}.q_i[B_i]; C_i)} [C]_{\text{PSTART}}
\end{array}$$

Figure 3: Choreography calculus, semantics.

$\tilde{r}$  ( $k'$  and  $\tilde{r}$  replace respectively  $k$  and  $\tilde{q}$  in the continuation). Intuitively, in the premise  $D, \delta \triangleright D'$  for  $\delta = \text{start } k : \tilde{l}.p[A]$ , we obtain  $D'$  from  $D$  adding the needed information to support the newly created session in the continuation  $C'$ . In particular, we need to add information about the location of each new process and to set up the state and queue maps of each process involved in the new session to enable communications via message correlation. We formalise this information in the auxiliary predicate of *session support*,  $\text{sup}$ :

**DEFINITION 2** (Session Support ( $\text{sup}$ )). *We say that  $(t, \{M_A\}_{A \in \tilde{A}})$  is a session support for a deployment  $D$  and some located roles  $\tilde{l}.A$ , denoted by the predicate  $\text{sup}(t, \{M_A\}_{A \in \tilde{A}}, D, \tilde{l}.A)$ , if and only if:*

- (Locations) *For all  $\tilde{l}.A \in \tilde{l}.A$ ,  $\mathbf{A}.l(t) = \tilde{l}$ .*
- (Correlation Keys) *For all pairwise-distinct  $\tilde{l}.A, \tilde{l}'.B$  in  $\tilde{l}.A$ ,  $\mathbf{A}.B(t) = t_c$  and  $M_B(t_c) = \varepsilon$  for some  $t_c$  such that  $t_c \notin \text{dom}(D(s).\text{que})$  for all  $s \in D(l')$ .*

The predicate  $\text{sup}(t, \{M_A\}_{A \in \tilde{A}}, D, \tilde{l}.A)$  holds if the tree  $t$ , from now on called *session descriptor*, contains the locations of the processes playing the roles in the session and the correlation keys that they use. Note that in  $t$  we use roles (which are static names) instead of process identifiers because in the message correlation mechanism found in SOC, a sender does not know the process identifier of the intended receiver (this will be reflected in the development of our target language, DCC, in § 5). The set  $\{M_A\}_{A \in \tilde{A}}$  contains the respective queue map  $M_A$  for each role  $A$ .  $M_A$  must contain an empty queue correlating with the corresponding key in  $t$  for each other role in the session, e.g., if role  $A$  receives from  $B$ ,  $M_A$  contains an empty queue correlating with the key at  $\mathbf{B}.A$  in  $t$ . Thus, each role has a queue to receive messages from each other role, as in standard multiparty session types with session-role channels [4]. The correlation keys must be fresh within the location of the receiver, i.e., none of the other processes running at that location use the same key to correlate<sup>6</sup>. We can now define the  $\triangleright$  relation for session start. Below,  $M \sqcup M'$  denotes the disjoint union of two queue maps ( $M$  and  $M'$  do not share correlation keys).

**DEFINITION 3** (Session Start). *Let  $\delta = \text{start } k : \tilde{l}.p[A]$ . Then  $D, \delta \triangleright D'$  if and only if the following conditions hold, for  $t$  and  $\{M_A\}_{A \in \tilde{A}}$  such that  $\text{sup}(t, \{M_A\}_{A \in \tilde{A}}, D, \tilde{l}.A)$*

- (Locations)  $\forall l, D'(l) = \begin{cases} D(l) \uplus \{q\} & \text{if } l.q[B] \in \tilde{l}.q[B] \text{ and } l \in \text{dom}(D) \\ \{q\} & \text{if } l.q[B] \in \tilde{l}.q[B] \text{ and } l \notin \text{dom}(D) \\ D(l) & \text{otherwise} \end{cases}$

<sup>6</sup> In AC, like in SOC, the sender delivers a message if it can find a process at the location of the receiver with a queue correlating with its key. Thus, to make correlation deterministic, keys must be unique within their locations.

- (States) *Let  $D(p_1) = (t_{p_1}, M_{p_1})$ ,  $\forall q, D'(q) = \begin{cases} (t_p \triangleleft (\underline{k}, t), M_p \sqcup M_A) & \text{if } q = p_1 \\ (t_\perp \triangleleft (\underline{k}, t), M_B) & \text{if } q[B] \in \tilde{l}.q[B] \\ D(q) & \text{otherwise} \end{cases}$*

Note that, according to the previous definition, we copy the session descriptor under  $\underline{k}$ , the path named after the session. This makes easier the access to the structure, since all in-session interactions happen under a certain session name  $k$  which is globally fresh and therefore cannot clash with other pre-existing structures. Observe also that we choose the session descriptor  $t$  and queue maps  $\{M_A\}_{A \in \tilde{A}}$  in a non-deterministic way, as there are potentially many that respect predicate  $\text{sup}$ . We made this choice to obtain a general model that allows for different implementations (e.g., based on cookies, random sets of data, API keys, databases, etc.), as long as they comply with our definition of session support (see § 7).

**REMARK 2.** *We could optimise Definition 3 to give a separate session descriptor to each process, removing unused data like its own location or correlation keys it does not use. However, we chose our definition because it is simpler and does not alter our result.*

**Rule  $[C]_{\text{SEND}}$ .** The rule describes a partial sending and, as shown in Figure 3, it is essentially a modification of the deployment to take into account asynchronous message passing.

**DEFINITION 4** (Session Send). *Assume  $\delta = k : p[A].e \rightarrow B.o$  and let  $D, D'$  be deployments. Then  $D, \delta \triangleright D'$  holds if and only if:*

- $l = \mathbf{k}.B.l(D(p).\text{st}) \wedge q \in D(l)$
- $t_c = \mathbf{k}.A.B(D(p).\text{st}) \wedge D(q).\text{que}(t_c) = \tilde{m}$
- $t_m = \text{eval}(e, D(p).\text{st})$
- $D' = D[q \mapsto (D(q).\text{st}, D(q).\text{que}[t_c \mapsto \tilde{m} :: (o, t_m)])]$

We comment the conditions of the effect referring paths as applied on the state of the sender  $p$ . The first two conditions find the receiving process  $q$  and its queue.  $q$  is that process  $i$  located in  $D$  at the location  $l$  at  $\mathbf{k}.B.l$ , and  $ii$ ) owning a queue correlating with the key at  $\mathbf{k}.A.B$ . This models real-world message correlation, which guesses the receiver from its location and the correlation key used by the sender.  $t_m = \text{eval}(e, D(p).\text{st})$  is the content of the message, result of the evaluation of the expression  $e$  on the state of  $p$ . The effect of  $[D]_{\text{SEND}}$  is that in  $D'$  the receiver  $q$  stores in the queue correlating with the key at  $\mathbf{k}.A.B$  the message  $(o, t_m)$ .

**Rule  $[C]_{\text{RECV}}$ .** Rule  $[C]_{\text{RECV}}$  implements a partial reception and, similarly to  $[C]_{\text{SEND}}$ , is basically a modification of the deployment. In particular, the rule records on which of the available operations ( $o_i, i \in I$ )  $q$  received a message from the process playing role  $A$ .

**DEFINITION 5** (Session Receive). *Let  $\delta = k : A \rightarrow q[B].o(x)$  and  $D, D'$  be deployments. Then  $D, \delta \triangleright D'$  holds if and only if:*

- $t_c = \mathbf{k}.A.B(D(q).\text{st}) \wedge D(q).\text{que}(t_c) = (o, t_m) :: \tilde{m}$
- $D' = D[q \mapsto (D(q).\text{st} \triangleleft (x, t_m), D(q).\text{que}[t_c \mapsto \tilde{m}])]$

Above, the first condition finds the proper queue that  $q$ , playing  $B$ , uses to receive from  $A$ . The second one provides a new deployment  $D'$  if the head of the queue has a message on operation  $o$ . If it does,  $D'$  copies in the state of  $q$ , under path  $x$ , the content of the message  $t_m$ .  $D'$  also removes the message from the head of the queue.

**Other rules.** Rule  $[C]_{\text{COM}}$  describes a complete communication. Notably, the effect  $\delta$  is the same of rule  $[C]_{\text{SEND}}$  and the continuation is a partial reception with only one branch on operation  $o$ . Thanks to the deployment, we can model asynchronous communication also in the complete case in a fairly simple way (wrt other approaches [8, 27]). As defined in Rule  $[C]_{\text{COM}}$ , we can store in the deployment the send “part” of the communication and transform the complete term into a partial reception for later execution. Rules  $[C]_{\text{COND}}$ ,  $[C]_{\text{CTX}}$ , and  $[C]_{\text{PAR}}$  are standard, while rule  $[C]_{\text{EQ}}$  accounts for the structural congruence  $\equiv$  and the swapping relation  $\simeq_C$ . The structural congruence, defined as the smallest congruence supporting  $\alpha$ -conversion and satisfying the rules below

$$C \mid C' \equiv C' \mid C \quad (C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3)$$

$$\text{def } X = C' \text{ in } 0 \equiv 0 \quad \text{def } X = C' \text{ in } C[X] \equiv \text{def } X = C' \text{ in } C[C']$$

allows to abstract from purely syntactic differences in processes and also to treat recursion in a standard way. As in [8], the swap relation  $\simeq_C$ , allows to swap the order of some actions. This allows more interleaving among processes. For instance, the Rule below swaps  $\eta$  and  $\eta'$  if they share no processes (returned by  $\text{pn}(\eta)$ ).

$$\text{pn}(\eta) \cap \text{pn}(\eta') = \emptyset \Rightarrow \eta; \eta' \simeq_C \eta'; \eta \quad [C]_{\text{ETAETA}}$$

Swapping instructions means that, even if a choreography defines a global order in which its processes shall send and receive their messages, this order can change at runtime. Despite this global change, we guarantee to preserve the order of messages between each couple of processes in a session. Rule  $[C]_{\text{PSTART}}$  starts a new session by synchronising a partial choreography that requests to start a session with other choreographies that can accept the request. The premise of the rule  $\{L.B\} = \biguplus_i \{l_i.B_i\}$ , where  $\biguplus$  indicates the disjoint union of the list of located roles, requires that in the accepting choreographies the list of locations and their supported roles match the corresponding list of the request. The rest of the rule is similar to  $[C]_{\text{START}}$ . The choreographies accepting the request remain available afterwards, for reuse.

### 2.3 An example

We show, as a comprehensive example, a verified file transfer system written in AC. We use the example in other sections to explain our endpoint projection and compilation.

**Verified file transfer.** The program *i*) validates the file request of a Client on a Server, *ii*) transfers a file in multiple parts, and *iii*) verifies the transfer with a checksum. The Server logs all requests.

We report in Figure 4 the code of program  $C$ , parallel composition of the two partial choreographies  $\bar{C} = C_c \mid C_s$ .  $C_c$  and  $C_s$  respectively define the client- and the server-side code.

In  $C$ , process  $c$  — the only active process — plays the role of the Client  $C$ . On the server-side,  $a$  plays the Access Manager ( $A$ ),  $dm$  the Download Manager ( $DM$ ) and  $l$  the Logger ( $L$ ). On the client-side (Line 18 of  $C_c$ ) process  $f$  plays  $F$ , accessing the file system. We conveniently locate  $A$ ,  $DM$ ,  $L$ , and  $F$  at respectively  $l_A$ ,  $l_{DM}$ ,  $l_L$ , and  $l_C$ .

Lines 1 of  $C_c$  and  $C_s$  start session  $k_d$  between  $c$ ,  $a$ ,  $dm$ , and  $l$ . At Lines 2 of  $C_c$  and  $C_s$ ,  $c$  makes a request for a file to  $a$ . Lines 4-15 of  $C_s$  define the outcome of a valid request (Line 3). Following the Lines,  $a$  forwards the resource request to  $dm$  and confirms (on *ok*) to  $c$  the request.  $dm$  asks  $l$  to log the request. Lines 7-14 define the recursive procedure **TRANSFER**, called at Line 15, for the multi-part file download. If  $dm$  has more packets of the resource, it sends the next to  $c$  on operation *pkt* and **TRANSFER** repeats. Else  $dm$  ends  $k_d$  sending to  $c$  the checksum of the file. Lines 17-19 of  $C_s$  specify the outcome of an invalid request:  $a$  notifies  $dm$  and  $c$  of the failed attempt (*ko*) and  $dm$  asks to log the event to  $l$ .

Observe that  $C_c$  defines at Lines 3-15 a procedure to **STORE** the file. At Line 18 of  $C_c$ , after the request approval,  $c$  (playing

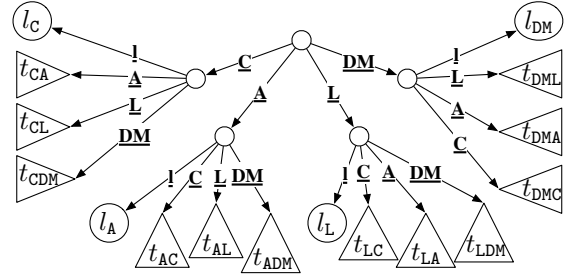


Figure 5: Example of structure of Session Descriptor

user  $U$ ) starts a new session  $k_s$  with  $f$  and it asks  $f$  to create a file to stores the incoming packets. Then, in **STORE**, if  $c$  receives a new packet of bytes from  $dm$  (Line 5), it asks  $f$  to append them to the local file and **STORE** repeats. Else  $c$  receives the checksum from  $dm$  (Line 8) and  $c$  forwards it to  $f$ . Finally,  $f$  notifies  $c$  whether it *saved* or *discarded* the file according to the checksum.

We comment some features of AC programs in the example.

**Session Descriptors.** At Lines 1 of  $C_c$  and  $C_s$   $c$  requests to start a new session  $k_d$  with some newly created processes  $a$ ,  $dm$  and  $l$ . Here, we comment the structure of the session descriptor included under path  $k_d$  in the state of the involved processes after the start.

We report in Figure 5 such structure. Following the path  $A$ , we find a subtree with *i*) a node  $l$  that stores the location  $l_A$  of  $a$  and *ii*) a set of nodes, named after all the other roles in  $k$ , leading to subtrees containing the correlation keys related to the roles in the paths. For example, the path  $A.C$  allows to reach the tree  $t_{AC}$ , which contains the correlation key used by  $a$  to send messages to  $c$ .

**Parallelism.** Lines 5-6 of  $C_s$  show how Rule  $[C]_{\text{EQ}}$  can exchange the order of execution of actions. The actions at Lines 5 and 6 regard different processes (resp.  $a$ ,  $s$ , and  $l$ ) and the two instructions can swap along rule  $[C]_{\text{ETAETA}}$  of the swap relation. A possible reduction of  $C$ , starting from Line 5 of  $C_s$ , can apply rule  $[C]_{\text{EQ}}$  to swap Lines 5 and 6, reduce Line 6 with Rule  $[C]_{\text{COM}}$  to a partial reception for process  $l$ , and swap back the two Lines. Next, either  $a$  delivers its message with  $[C]_{\text{SEND}}$  or  $[C]_{\text{EQ}}$  applies, swapping again the two Lines to let  $l$  consume its message. Observe that the swap is non-deterministic, allowing for other possible executions.

**Asynchrony.** AC supports asynchronous communication with queues, however, to achieve asynchrony, we need the swap relation to let a process send or receive a message, although its choreography defines that other actions should happen before it. Consider Lines 4-5 of  $C_s$ . We can apply  $[C]_{\text{COM}}$  on Line 4, letting  $a$  send its message to  $dm$  and reducing Line 4 to a partial reception on  $dm$ . Then, we can apply  $[C]_{\text{EQ}}$ , swapping the redex of Line 4 with Line 5 and let Line 5 execute with  $[C]_{\text{SEND}}$ . Finally, we apply  $[C]_{\text{RECV}}$  on the redex of Line 4 and let  $dm$  consume its message.

## 3. Typing

In this section we define our typing discipline for Applied Choreographies, which checks the behaviour of sessions against protocols given as multiparty session types [13, 17]. The main novelty wrt previous type systems for choreographies is checking that the evolution of a deployment (states for message correlation and queues for asynchronous messaging) correctly implements the sessions described in a program, ensuring absence of errors such as deadlocks. We explore this part in detail in § 3.3.

### 3.1 Types and Type Projection

**Global and Local types.** As in standard multiparty session types, we use *global types* to represent protocols from a global viewpoint and *local types* to describe the behaviour of each participant. Our type system checks that the local types that abstract the behaviour of processes in a choreography coherently follow a global type. The

$$C_c = \left\{ \begin{array}{l} 1. \text{ req } k_d : c[C] \Leftrightarrow l_A.A, l_{DM}.DM, l_L.L; \\ 2. k_d : c[C].mkReq() \rightarrow A.get; \\ 3. \text{ def STORE} = \\ 4. \quad k_d : DM \rightarrow c[C].\{ \\ 5. \quad \quad pkt(\underline{bb}); \\ 6. \quad \quad k_s : c[U].\underline{bb} \rightarrow f[F].append(\underline{data}); \\ 7. \quad \quad \text{STORE}, \\ 8. \quad \quad chksum(\underline{cs}); \\ 9. \quad \quad k_s : c[U].\underline{cs} \rightarrow f[F].check(\underline{cs}); \\ 10. \quad \quad \text{if } f.check(\underline{fname}, \underline{cs}) \{ \\ 11. \quad \quad \quad k_s : f[F] \rightarrow c[U].saved \\ 12. \quad \quad \quad \} \text{ else } \{ \\ 13. \quad \quad \quad k_s : f[F] \rightarrow c[U].discarded \\ 14. \quad \quad \} \} \\ 15. \text{ in} \\ 16. k_d : A \rightarrow c[C].\{ \\ 17. \quad ok; \\ 18. \quad \text{start } k_s : c[U] \Leftrightarrow l_C.f[F]; \\ 19. \quad k_s : c[U].name() \rightarrow f[F].create(\underline{fname}); \\ 20. \quad \text{STORE}, \\ 21. \quad ko \} \end{array} \right.$$

$$C_s = \left\{ \begin{array}{l} 1. \text{ acc } k_d : l_A.a[A], l_{DM}.dm[DM], l_L.l[L]; \\ 2. k_d : C \rightarrow a[A].get(\underline{req}); \\ 3. \text{ if } a.isValid(\underline{req}) \{ \\ 4. \quad k_d : a[A].req.rsc \rightarrow dm[DM].ok(\underline{rsc}); \\ 5. \quad k_d : a[A] \rightarrow C.ok; \\ 6. \quad k_d : dm[DM].logok(\underline{rsc}) \rightarrow l[L].log(\underline{log}); \\ 7. \quad \text{def TRANSFER} = \\ 8. \quad \quad \text{if } dm.more(\underline{rsc}) \{ \\ 9. \quad \quad \quad k_d : dm[DM].next(\underline{rsc}) \rightarrow C.pkt; \\ 10. \quad \quad \quad \text{TRANSFER} \\ 11. \quad \quad \quad \} \text{ else } \{ \\ 12. \quad \quad \quad k_d : dm[DM].chksum(\underline{rsc}) \rightarrow C.chksum \\ 13. \quad \quad \quad \} \\ 14. \quad \text{in} \\ 15. \quad \text{TRANSFER} \\ 16. \quad \} \text{ else } \{ \\ 17. \quad k_d : a[A].req.rsc \rightarrow dm[DM].ko(\underline{rsc}); \\ 18. \quad k_d : a[A] \rightarrow C.ko; \\ 19. \quad k_d : dm[DM].logko(\underline{rsc}) \rightarrow l[L].log(\underline{log}) \\ 20. \quad \} \end{array} \right.$$

Figure 4: Choreography Example

syntax of global types  $G$  and local types  $T$  is reported in Figure 6. A global type  $A \rightarrow B.\{o_i(U_i); G_i\}_i$  abstracts a communication, where  $A$  can send to  $B$  a message on any of the operations  $o_i$  and continue with the respective continuation  $G_i$ . A carried type  $U$  types the tree value exchanged in the message. Specifically, a tree type  $S\{\underline{x}_i : U_i\}_i$  abstracts a tree with root value  $S$  (a basic type) and subtrees reachable from the root node by following  $\underline{x}_i$  with respective types  $U_i$  (our notation recalls that for record types in [32]). In local types,  $!A.\{o_i(U_i); T_i\}_i$  abstracts the sending of a message of type  $U_i$  to role  $A$  on one of the operations  $o_i$ , with continuation  $T_i$ . Dually,  $?A.\{o_i(U_i); T_i\}_i$  abstracts the offering of an input choice for all the operations  $o_i$ , with continuation  $T_i$ . All other terms for recursion and termination (end) are standard.

$$\begin{aligned} G &::= A \rightarrow B.\{o_i(U_i); G_i\}_i \mid \text{rec } t; G \mid t \mid \text{end} \\ \alpha &::= !A \mid ?B \quad T ::= \alpha.\{o_i(U_i); T_i\}_i \mid \text{rec } t; T \mid t \mid \text{end} \\ U &::= S\{\underline{x}_i : U_i\}_i \quad S ::= \text{int} \mid \text{bool} \mid \text{str} \mid \dots \end{aligned}$$

Figure 6: Global and Local Types.

**Type Projection.** To relate global types to the behaviour of endpoints, we project a global type  $G$  onto the local type of a single role. We report in Figure 7 the projection of global types, which we define following [27]. We write  $\llbracket G \rrbracket_A$  to denote the projection of  $G$  onto the role  $A$ . Intuitively,  $\llbracket G \rrbracket_A$  gives an encoding of the local actions expected by role  $A$  in the global type  $G$ . When projecting a communication we require the local behaviour of all roles not involved in it to be merged with the merging operator  $\sqcup$ . Like in [27]  $T \sqcup T'$  is isomorphic to  $T$  and  $T'$  up to branching, where all branches of  $T$  or  $T'$  with distinct operations are also included.

### 3.2 Type checking

We now present our system to check that sessions in a choreography follow their types.

**Environments.** We define our typing environments  $\Gamma, \Gamma', \dots$  as:

$$\begin{array}{c} \Gamma ::= \Gamma, \tilde{l} : G\langle A \rangle \tilde{B} \tilde{C} \mid \Gamma, k[A] : T \mid \Gamma, p : k[A] \\ \mid \Gamma, p.x : U \mid \Gamma, X : \Gamma \mid \emptyset \end{array}$$

A service typing  $\tilde{l} : G\langle A \rangle \tilde{B} \tilde{C}$  types with  $G$  all sessions created by contacting the services at the locations  $\tilde{l}$ . We explain the role annotations:  $A$  is the role that the active process (the starter) should play; roles  $\tilde{B}$  are the roles respectively played by each  $l$  in  $\tilde{l}$  (each  $l$  plays one role, so the length of  $\tilde{B}$  is the same as the length of  $\tilde{l}$ );

$$\begin{aligned} \llbracket t \rrbracket_A &= t \quad \llbracket \text{end} \rrbracket_A = \text{end} \quad \llbracket \text{rec } t; G \rrbracket_A = \begin{cases} \text{rec } t; \llbracket G \rrbracket_A & \text{if } A \in G \\ \text{end} & \text{otherwise} \end{cases} \\ \llbracket A \rightarrow B.\{o_i(U_i); G_i\}_i \rrbracket_C &= \begin{cases} !B.\{o_i(U_i); \llbracket G_i \rrbracket_C\}_i & \text{if } C = A \\ ?A.\{o_i(U_i); \llbracket G_i \rrbracket_C\}_i & \text{if } C = B \\ \sqcup_i \llbracket G_i \rrbracket_A & \text{otherwise} \end{cases} \end{aligned}$$

Figure 7: Choreography Calculus - Global Type Projection

finally,  $\tilde{C}$  (where  $\tilde{C} \subseteq \tilde{B}$ ) are the roles implemented by the choreography that we are typing. The annotation  $\tilde{C}$  enables choreographies to be composed and forbids multiple choreographies to declare the implementation of the same roles, as in [27]. When we write  $\Gamma, \tilde{l} : G\langle A \rangle \tilde{B} \tilde{C}$ , we always assume that: *i)*  $\{A, \tilde{B}\} = \text{roles}(G)$ , where roles returns the set of roles in  $G$ ; *ii)* the locations  $\tilde{l}$  are ordered lexicographically; and, *iii)* the locations in  $\tilde{l}$  do not appear in any other service typing in  $\Gamma$ . A local typing  $k[A] : T$  states that role  $A$  in session  $k$  follows the local type  $T$ . We assume roles in a session are typed by a single local typing, as in standard multiparty session types [17]. An ownership typing  $p : k[A]$  states that process  $p$  owns the role  $A$  in session  $k$ : each process is allowed to participate in multiple sessions, but to play only one role in each of such sessions. Hence, a process  $p$  may appear in more than one ownership typings in a  $\Gamma$ , but never more than once per session. The other typings for variables and recursive procedures are standard.

**Typing Judgements and Rules.** A judgement  $\Gamma \vdash C$  states that the choreography  $C$  follows the specifications given in  $\Gamma$ . We report in Figure 8 the typing rules to derive valid typing judgements.

We comment the rules. Rule  $[T]_{\text{START}}$  types a session start. In the first premise, the service typing  $\tilde{l} : G\langle A \rangle \tilde{B} \tilde{B}$  checks that the continuation implements all the roles in protocol  $G$ . The auxiliary function  $\text{init}^7$  intuitively returns an environment containing all the ownerships and local typings to correctly type a freshly-started session. The type of each process is the local type projection of the global type  $G$  on the role owned by the process in the session.

Rule  $[T]_{\text{REQ}}$  is similar to  $[T]_{\text{START}}$ , but performs the checks only for the process requesting the creation of a new session. Dually,  $[T]_{\text{ACC}}$  checks that the processes created by receiving a request correctly implement their expected behaviour.

Rule  $[T]_{\text{COM}}$  types a complete communication, checking that: *i)* the chosen operation  $o_j$  is among the ones that the sender can select according to its local type; *ii)* similarly,  $o_j$  is among the ones

<sup>7</sup>Formally  $\text{init}(\tilde{p}[A], k, G) = \{q : k[B], k[B] : \llbracket G \rrbracket_B \mid q[B] \in \tilde{p}[A]\}$ .

$$\begin{array}{c}
\frac{\Gamma, \tilde{l} : G\langle A|\tilde{B}|\tilde{C} \rangle, \text{init}(\widetilde{r[C]}, k, G) \vdash C \quad \widetilde{r[C]} = p[A], \widetilde{q[B]} \quad \tilde{q} \notin \Gamma}{\Gamma, \tilde{l} : G\langle A|\tilde{B}|\tilde{C} \rangle \vdash \text{start } k : p[A] \Leftrightarrow \tilde{l}.q[\tilde{B}]; C} [T]_{\text{START}} \\
\\
\frac{\Gamma, p : k[A], k[A] : [G]_A \vdash C \quad \Gamma \vdash \tilde{l} : G\langle A|\tilde{B}|\emptyset \rangle}{\Gamma \vdash \text{req } k : p[A] \Leftrightarrow \tilde{l}.B; C} [T]_{\text{REQ}} \quad \frac{\tilde{l} \subseteq \tilde{l}' \quad \Gamma, \tilde{l}' : G\langle A|\tilde{B}|\emptyset \rangle, \text{init}(\widetilde{q[C]}, k, G) \vdash C}{\Gamma, \tilde{l}' : G\langle A|\tilde{B}|\tilde{C} \rangle \vdash \text{acc } k : \tilde{l}.q[\tilde{C}]; C} [T]_{\text{ACC}} \\
\\
\frac{\Gamma \vdash p.e : \text{bool} \quad \Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash \text{if } p.e \{C_1\} \text{ else } \{C_2\}} [T]_{\text{COND}} \quad \frac{j \in I \quad \Gamma \vdash p : k[A], q : k[B] \quad \Gamma \vdash p.e : U_j \quad \Gamma, q.x : U_j, k[A] : T_j, k[B] : T'_j \vdash C}{\Gamma, k[A] : !B.\{o_i(U_i); T_i\}_{i \in I}, k[B] : ?A.\{o_i(U_i); T'_i\}_{i \in I} \vdash k : p[A].e \rightarrow q[B].o_j(x); C} [T]_{\text{COM}} \\
\\
\frac{j \in I \quad \Gamma \vdash p : k[A] \quad \Gamma \vdash p.e : U_j \quad \Gamma, k[A] : T_j \vdash C}{\Gamma, k[A] : !B.\{o_i(U_i); T_i\}_{i \in I} \vdash k : p[A].e \rightarrow B.o_j; C} [T]_{\text{SEND}} \quad \frac{\Gamma \vdash q : k[B] \quad \forall j \in I. \Gamma, q.x_j : U_j, k[B] : T_j \vdash C_j}{\Gamma, k[B] : ?A.\{o_i(U_i); T_i\}_{i \in I} \vdash k : A \rightarrow q[B].\{o_j(x_j); C_j\}_{j \in I \cup J}} [T]_{\text{RECV}} \\
\\
\frac{\Gamma_i \vdash C_i \quad \Gamma_2 \vdash C_2}{\Gamma_1 \circ \Gamma_2 \vdash C_1 \mid C_2} [T]_{\text{PAR}} \quad \frac{\text{end}(\Gamma)}{\Gamma \vdash \mathbf{0}} [T]_{\text{END}} \quad \frac{\Gamma, X : \Gamma' \vdash C \quad \Gamma', X : \Gamma' \vdash C' \quad \Gamma'_{|\text{locs}} \subseteq \Gamma \quad \tilde{p} = \text{pn}(C')}{\Gamma \vdash \text{def } X(\tilde{p}) = C' \text{ in } C} [T]_{\text{DEF}} \quad \frac{\text{end}(\Gamma) \quad \Gamma'' \subseteq \Gamma'}{\Gamma, \Gamma'', X(\tilde{p}) : \Gamma' \vdash X(\tilde{p})} [T]_{\text{CALL}}
\end{array}$$

Figure 8: Choreography Calculus - Typing Rules

offered by the receiver according to its local type; *iii*) the sender and the receiver processes own their respective roles in the session; *iv*) the expression of the sender ( $e$ ) has the type<sup>8</sup>  $U_j$  expected by the protocol; *v*) the receiver uses the reception variable accordingly in the continuation  $C$ ; and *vi*) processes  $p$  and  $q$  proceed according to their respective types in  $\Gamma$ . Similar to rule  $[T]_{\text{COM}}$ ,  $[T]_{\text{SEND}}$  and  $[T]_{\text{RECV}}$  respectively check (*send*) and (*recv*) actions.

Rule  $[T]_{\text{PAR}}$  uses the *role distribution operator*  $\Gamma_1 \circ \Gamma_2$ , from [27], to check that choreographies executing in parallel do not implement overlapping roles at locations. Formally,  $\Gamma_1 \circ \Gamma_2$  is defined as, let  $\text{pn}(\Gamma_1) \cap \text{pn}(\Gamma_2) = \emptyset^9$  and  $\Gamma_i = \Gamma'_i, \Gamma_i^l, i \in \{1, 2\}$  where  $\Gamma_i^l$  contains only service typings

$$\Gamma_1 \circ \Gamma_2 = \Gamma'_1, \Gamma'_2, \Gamma_1^l \circ \Gamma_2^l \\
\Gamma_1^l \circ \Gamma_2^l = \left\{ \tilde{l} : G\langle A|\tilde{B}|\tilde{C} \rangle \mid \begin{array}{l} \tilde{l} : G\langle A|\tilde{B}|\tilde{D} \rangle \in \Gamma_1^l \wedge \\ \tilde{l} : G\langle A|\tilde{B}|\tilde{E} \rangle \in \Gamma_2^l \wedge \tilde{D} \uplus \tilde{E} = \tilde{C} \end{array} \right\}$$

All the other typing rules are standard. In Rule  $[T]_{\text{END}}$  *end* holds if the protocols for all sessions have terminated (i.e., all local typings have type *end*). In Rule  $[T]_{\text{DEF}}$  the condition  $\Gamma'_{|\text{locs}} \subseteq \Gamma$  checks that the body of the recursive procedure does not introduce unexpected services ( $\Gamma'_{|\text{locs}}$  returns all service typings in  $\Gamma'$ ).

### 3.3 Runtime Typing

To prove that well-typed AC programs never go wrong, we need to pay attention to how their deployments evolve at runtime. For example, in Rule  $[C]_{\text{COM}}$  the sender needs the necessary information in its state to “find” the receiver through correlation. This is a remarkable difference wrt previous works on choreographies, where such conditions do not exist and therefore choreographies can always continue execution (see, e.g., [7–9, 34]). To address this issue, we extend our typing discipline to check runtime states.

**Wrong Deployments.** We need to prevent deployments from “going wrong” during execution. Intuitively, we say that a deployment is wrong wrt a choreography if any of these conditions holds:

- (*uninitialised variables*) processes have undefined variables;
- (*incompatible session descriptors*) processes in a session store different locations or keys for the same session descriptor;
- (*correlation race*) a correlation key is used for more than one queue at a single location;
- (*protocol violations*) a message queue does not contain messages as expected by the protocol of the session it is used for.

Wrong deployments may cause unpredictable executions or undesired behaviours, e.g., deadlocks. We illustrate the consequences of

having wrong deployments with this simple running choreography:

$$D, k : p[A].y \rightarrow q[B].x; \mathbf{0}$$

The only way this choreography can reduce is to apply Rule  $[C]_{\text{COM}}$  in a reduction derivation. The second premise of Rule  $[C]_{\text{COM}}$  requires a deployment reduction of the form  $D, k : p[A].y \rightarrow B \blacktriangleright D'$  for some  $D'$ . Hence, the deployment  $D$  must respect the conditions given in Definition 4. Let us see how a wrong deployment  $D$  may cause problems, following the above list:

- (*uninitialised variables*) Assume that  $D$  is such that  $D(p).\text{st}$  is a tree with no node under path  $y$ ; then the condition  $\text{eval}(y, D(p).\text{st})$  given in Definition 4 is undefined and  $[C]_{\text{COM}}$  cannot be applied, causing the choreography to get stuck.
- (*incompatible session descriptors*) Assume  $q \in D(l')$  for  $l \neq l'$  and  $k.A.B.l(D(p).\text{st}) = l$ . Again, Definition 4 cannot apply and we have a deadlock, caused by the sender “pointing” at the wrong receiving service. We have a similar case also if we assume  $t \notin \text{dom}(D(q).\text{que})$  and  $k.A.B(D(p).\text{st}) = t$  because we cannot find the queue of the addressee.
- (*correlation race*) Assume  $D$  is such that  $D(l) = \{q, r\}$  ( $r$  and  $q$  are at the same location). Assume also  $D(q).\text{que} = D(r).\text{que} = M$  for some  $M$  such that  $k.A.B(D(p).\text{st}) \in M$ , i.e., both  $q$  and  $r$  have a queue correlating with the key that  $p$  uses to send its message. Since Definition 4 guesses the receiver from its location and correlation key,  $p$  non-deterministically delivers its message to either  $q$  or  $r$ . In the second case, we get:

$$D, k : p[A].y \rightarrow q[B].o(x); \mathbf{0} \rightarrow D', k : A \rightarrow q[B].x; \mathbf{0}$$

where in  $D'$  the queue of  $r$  contains the message received from  $p$ . The choreography is now deadlocked because  $q$  cannot consume the expected message from  $p$ .

- (*protocol violations*) Assume that  $D(q).\text{que} = M$  for some  $M(k.A.B(D(p).\text{st})) = (o', t')$  where  $o \neq o'$ , i.e.,  $q$  has a message in the queue used by  $p$ . If we let the choreography reduce like in the previous point, it ends up deadlocked. After the reduction, the queue used by  $p$  still contains in its head the message  $(o', t')$  and Rule  $[C]_{\text{RECV}}$  cannot apply because it expects to find a message for  $o$  at that position.

Below we extend our type system to prove that, given a well-typed choreography, our semantics never produces wrong deployments (provided that we do not start from a wrong deployment). Observe that this development is transparent to programmers, since default deployments are never wrong.

**Runtime Global Types.** We extend the syntax of global types to capture partial runtime states, following the idea presented in [14]:

$$G ::= \dots \mid A \rightsquigarrow B.o(U); G$$

<sup>8</sup> The judgement  $\vdash t : U$  reads as “tree  $t$  has type  $U$ ”.

<sup>9</sup>  $\text{pn}(\Gamma) = \{p \mid p : k[A] \in \Gamma\}$

where the new term  $A \rightsquigarrow B.o(U)$  means that the sender  $A$  has sent the message but the receiver  $B$  has still to consume it.

**Semantics of Global Types.** To express the (abstract) execution of protocols, we give a semantics for global types. Formally,  $G \rightarrow G'$  is the smallest relation on the recursion-unfolding of global types satisfying the rules below

$$\begin{array}{l} \begin{array}{c} [G]_{\text{SEND}} \\ [G]_{\text{RECV}} \\ [G]_{\text{REC}} \\ [G]_{\text{SWAP}} \end{array} \quad \begin{array}{l} j \in I \Rightarrow A \rightarrow B.\{o_i(U_i); G_i\} \rightarrow A \rightsquigarrow B.o_j(U_j); G_j \\ A \rightsquigarrow B.o(U); G \rightarrow G \\ G[\text{rec } t; G/t] \rightarrow G' \Rightarrow \text{rec } t; G \rightarrow G' \\ G_1 \simeq_G G_2 \wedge G_2 \rightarrow G'_2 \wedge G'_2 \simeq_G G'_1 \Rightarrow G_1 \rightarrow G'_1 \end{array} \end{array}$$

The rules are similar to those for AC. Rule  $[G]_{\text{SEND}}$  allows a sender role to proceed before the corresponding receiver has actually received the message. Based on the selected operation, e.g.,  $o_j$ , the Rule reduces the type to a reception followed by the respective continuation ( $G_j$ ). The reception is executed in Rule  $[G]_{\text{RECV}}$ . In  $[G]_{\text{SWAP}}$ , we model parallelism with the relation for global types  $\simeq_G$ , which follows the same intuition of  $\simeq_C$ . Swapping allows us to model asynchrony in global types as done in AC. For example, we can swap a reception with a complete communication:

$$\frac{\{A, B\} \cap \{D\} = \emptyset}{\begin{array}{l} A \rightarrow B.\{o_i(U_i); C \rightsquigarrow D.o(U); G_i\} \\ \simeq_G C \rightsquigarrow D.o(U); A \rightarrow B.\{o_i(U_i); G_i\} \end{array}} [GS]_{\text{COMRECV}}$$

**Runtime Type checking and Typing Rules** We extend the typing rules given in the previous section to check runtime terms. The extension consists in *i*) new terms for  $\Gamma$  and *ii*) the introduction of rule  $[T]_{\text{DC}}$  to type runtime choreographies.

We extend the grammar of typing environments  $\Gamma$  as follows:

$$\Gamma ::= \dots \mid \Gamma, p@l \mid \Gamma, b[k]_B^A : T$$

We write  $p@l$  in  $\Gamma$  to state that process  $p$  runs at location  $l$ . A buffer typing  $b[k]_B^A : T$  types the messages currently in the queue used by the process implementing role  $B$  in session  $k$  to receive from role  $A$ .

To relate the buffer typings of queues used in a session with those expected by the protocol of such session, we define the *buffer type projection*  $\llbracket G \rrbracket_B^A$ , which returns the expected buffer type of role  $B$  from  $A$  in  $G$ .  $\llbracket G \rrbracket_B^A$  extracts from  $G$  the partial receptions of the form  $A \rightsquigarrow B.o(U)$ , translating it to a local type  $?A.o(U)$ . Below we report the formal definition of  $\llbracket G \rrbracket_B^A$  for the case of receptions.

$$\llbracket C \rightsquigarrow D.o(U); G \rrbracket_B^A = \begin{cases} ?A.o(U); \llbracket G \rrbracket_B^A & \text{if } C = A \wedge D = B \\ \llbracket G \rrbracket_B^A & \text{otherwise} \end{cases}$$

We also extend type projection to handle receptions:

$$\llbracket A \rightsquigarrow B.o(U); G \rrbracket_C = \begin{cases} ?A.o(U); \llbracket G \rrbracket_C & \text{if } C = B \\ \llbracket G \rrbracket_C & \text{otherwise} \end{cases}$$

We now proceed defining the *partial coherence* predicate  $\text{pco}(\Gamma)$ , which holds if and only if for all sessions  $k$ , the local and buffer typings of  $k$  follow (are projection of) the same global type  $G$ . The idea is that, since  $D$  is a global deployment, we can check for partial coherence of all sessions in Rule  $[T]_{\text{DC}}$ .

**DEFINITION 6 (Partial Coherence).** We write  $\text{pco}(\Gamma)$  when, for all sessions  $k$  in  $\Gamma$ , there exists a global type  $G$  such that,

$$\forall k[B] : T \in \Gamma \Rightarrow T = \llbracket G \rrbracket_B^A \wedge \forall A \in \text{roles}(G)/\{B\} \Rightarrow \Gamma \vdash b[k]_B^A : \llbracket G \rrbracket_B^A$$

Finally, we define the rule to type a running choreography:

$$\frac{\text{pco}(\Gamma) \quad \Gamma \vdash D \quad \Gamma \vdash C}{\Gamma \vdash D, C} [T]_{\text{DC}}$$

A judgement  $\Gamma \vdash D, C$  states that  $C$  and  $D$  are coherent according to  $\Gamma$ . The typing environment acts as an abstraction between  $D$  and  $C$  to guarantee that  $D$  will not go wrong. The premise  $\Gamma \vdash D$  checks  $D$  to be well-typed (not wrong) wrt to  $\Gamma$ . Formally:

**DEFINITION 7 (Deployment Judgements).**

$$\Gamma \vdash D \iff \left\{ \begin{array}{l} \forall l \in D, \forall p, q \in D(l), \\ \quad \text{dom}(D(p).\text{que}) \cap \text{dom}(D(q).\text{que}) = \emptyset \\ \forall p.x : U \in \Gamma, \vdash x(D(p).\text{st}) : U \\ \forall (p : k[A], q : k[B]) \in \Gamma, \\ \quad \underline{k}(D(p).\text{st}) = \underline{k}(D(q).\text{st}) \\ \forall p : k[A], \\ \quad \Gamma \vdash p@l \wedge p \in D(l) \wedge \underline{k.A.l}(D(p).\text{st}) = l \\ \forall p : k[A], \forall b[k]_A^B : T \in \Gamma, \\ \quad \text{bte}(B, D(p).\text{que}(\underline{k.B.A}(D(p).\text{st}))) = T \end{array} \right.$$

We comment, from top to bottom, the checks performed by  $\Gamma \vdash D$ :

- locations must have unique keys: for all locations in  $D$  and for all pairs of processes in that location, the queue maps of the two processes have no common correlation key (i.e., the domain of their queue maps are distinct).
- $\Gamma$  and  $D$  must agree on the type of variables: for each typing  $p.x : U$  in  $\Gamma$ ,  $D$  must associate  $x$ , in the state of process  $p$ , to a value of type  $U$ ;
- session descriptors must match: for all pair-wise distinct couples of processes  $p$  and  $q$ , playing the respective roles  $A$  and  $B$  in a session  $k$  in  $\Gamma$ , the session descriptors for  $k$  stored by  $p$  and  $q$  are the same;
- $\Gamma$  and  $D$  must agree on the location of all processes: for each process within a session  $k$ : *i*) its location according to  $\Gamma$ , *ii*) its location according to  $D$ , and *iii*) its location in the session descriptor of  $k$  must coincide;
- $\Gamma$  and  $D$  must agree on the state of all queues: for each process  $p$  playing role  $A$  in a session  $k$  and for each role  $B$  such that the buffer type  $b[k]_A^B : T \in \Gamma$ , the extracted buffer type of  $\underline{k.A.B}(D(p).\text{st})$  must be equal  $T$ . To check this last requirement, we define the *buffer typing extractor*  $\text{bte}(B, \tilde{m})$  which, given a queue  $\tilde{m}$  and a sender role  $B$ , returns the buffer type of  $\tilde{m}^{10}$ .

### 3.4 Properties

We close this section with the main guarantees of our type system.

First, our semantics preserves well-typedness:

**THEOREM 1 (Subject Reduction).**  $\Gamma \vdash D, C$  and  $D, C \rightarrow D', C'$  imply  $\Gamma' \vdash D', C'$  for some  $\Gamma'$ .

We now relate the behaviour of sessions in a well-typed choreography to their respective global types. We denote  $\llbracket G \rrbracket_k$  the projection of a global type  $G$  for a session  $k$  and let  $\llbracket G \rrbracket_k$  be the set of local and buffer typings as obtained by the projection of  $G$  on each of its roles:

**DEFINITION 8 (Global Type Projection).**

$$\llbracket G \rrbracket_k = \{k[A] : \llbracket G \rrbracket_A^A \mid A \in \text{roles}(G)\}, \\ \{b[k]_B^A : \llbracket G \rrbracket_B^A \mid A \in \text{roles}(G), B \in \text{roles}(G)/\{A\}\}$$

We say that a reduction is “at session  $k$ ” if it is obtained by consuming a communication term for session  $k$  (as in [17]), and we write  $k \notin \Gamma$  when  $k$  does not appear in any local typing in  $\Gamma$ . Then we have:

**THEOREM 2 (Session Fidelity).** Let  $\Gamma, \Gamma_k \vdash D, C, k \notin \Gamma$ . Then,  $D, C \rightarrow D', C'$  with a redex at session  $k$  implies that, for some  $G$  and  $\Gamma', k \notin \Gamma'$ , *(i)*  $\Gamma_k \subseteq \llbracket G \rrbracket_k$ , *(ii)*  $G \rightarrow G'$ , *(iii)*  $\Gamma_k \subseteq \llbracket G' \rrbracket_k$ , and *(iv)*  $\Gamma', \Gamma_k \vdash D', C'$ .

<sup>10</sup> Formally, let  $t_1 : U_1, \dots, t_n : U_n$  then

$\text{bte}(A, (o_1, t_1) :: \dots :: (o_n, t_n)) = ?A.o_1(U_1); \dots ; ?A.o_n(U_n)$



Theorem 2 states that all communications on sessions follow the expected protocols ( $\Gamma'$  may differ from  $\Gamma$  for the instantiation of a new variable).

We can now present one of our major results: well-typed applied choreographies never deadlock when all the necessary participants are defined. Let the coherence predicate  $\text{co}$  be defined as follows:

DEFINITION 9 (Coherence).  $\text{co}(\Gamma)$  holds iff  $\forall k \in \Gamma, \exists G$  s.t.

- $\tilde{l} : G\langle A|\tilde{B}|\tilde{C} \rangle \in \Gamma \wedge \tilde{C} = \tilde{B}$  and
- $\forall A \in \text{roles}(G),$ 
  - $k[A] : T \in \Gamma \wedge T = \llbracket G \rrbracket_A$  and
  - $\forall B \in \text{roles}(G)/\{A\}, b[k]_A^B = \llbracket G \rrbracket_A^B$

Coherence extends partial coherence to check that *i*) all the services needed to start new sessions are present and that *ii*) all the roles in every open session are correctly implemented by some processes. When a system is coherent and well-typed, it is also deadlock-free:

THEOREM 3 (Deadlock-freedom).  $\Gamma \vdash D, C$  and  $\text{co}(\Gamma)$  imply that either (i)  $C \equiv \mathbf{0}$  or (ii) there exist  $D'$  and  $C'$  such that  $D, C \rightarrow D', C'$ .

## 4. Endpoint Projection

We now present the Endpoint Projection and its properties. The EPP returns a correct composition<sup>11</sup> of endpoint choreographies that implements the behaviour of a given choreography. Intuitively, an endpoint (applied) choreography is an applied choreography that does not contain complete actions. Formally, let  $\text{fp}(C)$  return the set of free processes in a choreography  $C$ :

DEFINITION 10 (Endpoint Choreography [27]). We say that a choreography  $C$  is an endpoint choreography if  $C$  does not contain complete actions and one of the two following conditions holds:

- i*)  $C = \text{acc } k : l.q[B]; C' \wedge \{q\} = \text{fp}(C')$ ;    *ii*)  $\text{fp}(C) = \{p\}$

An endpoint choreography defines the identity of only one process: either *i*) a service process or *ii*) an active process. Remarkably, with this definition of EPP we can capture the description of the behaviour of single endpoints up to complete choreographies. To give the definition of the EPP of a choreography, we first define the notion of *process projection*, which defines the behaviour of a single process  $p$  in a choreography  $C$ , written  $\llbracket C \rrbracket_p$ .

Intuitively, the process projection follows the structure of the originating choreography. A (*start*) projects to a (*req*) on the active process and a set of (*acc*) terms on the service processes. Likewise, a (*com*) projects to a partial (*send*) for the sender and a partial (*recv*) for the receiver. Any partial term is projected as it is for its respective process, following the structure of the choreography. The only exceptions are (*recv*) and (*cond*) terms, whose projections are reported below, which use the *merging* partial operator  $\sqcup$  [9] to merge the behaviour of all processes in their branches.

$$\llbracket k : p[A].e \rightarrow q[B].o(x); C \rrbracket_r = \begin{cases} k : p[A].e \rightarrow B.o; \llbracket C \rrbracket_r & \text{if } r = p \\ k : A \rightarrow q[B].o(x); \llbracket C \rrbracket_r & \text{if } r = q \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases}$$

$$\llbracket \text{if } p.e \{C_1\} \text{ else } \{C_2\} \rrbracket_r = \begin{cases} \text{if } p.e \{ \llbracket C_1 \rrbracket_r \} \text{ else } \{ \llbracket C_2 \rrbracket_r \} & \text{if } r = p \\ \llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r & \text{otherwise} \end{cases}$$

$C \sqcup C'$  is defined only for endpoint choreographies and returns an endpoint choreography that is isomorphic to  $C$  and  $C'$  up to branching, i.e., it includes all branches on distinct operations. We report the only special rule of  $\sqcup$  for merging (*recv*) terms.

$$k : A \rightarrow p[B]. \{o_i(x_i); C_i\}_{i \in I} \sqcup k : A \rightarrow q[B]. \{o_j(x_j); C'_j\}_{j \in J} = k : A \rightarrow p[B]. \left\{ \begin{array}{l} \{o_i(x_i); C_i\}_{i \in I/J} \cup \{o_i(x_i); C'_i\}_{i \in J/I} \\ \cup \{o_i(x_i); C_i \sqcup C'_i\}_{i \in I \cap J} \end{array} \right\}$$

<sup>11</sup> Here and in the following “composition” means “parallel composition”.

On the definition of process projection, we can now define the EPP of a whole system. Here, and in the following,  $\llbracket C \rrbracket_l$  is the grouping operator that returns the set of grouped (bound) service processes at the same location. We show only the rules of  $\llbracket C \rrbracket_l$  for (*start*) and (*acc*) as the other rules are trivially recursive applications.

$$\begin{aligned} \llbracket \text{start } k : p[D] \Leftrightarrow l.q[B]; C \rrbracket_l &= \llbracket \text{acc } k : l.q[B]; C \rrbracket_l \\ \llbracket \text{acc } k : l.q[B]; C \rrbracket_l &= \begin{cases} \{r\} \cup \llbracket C \rrbracket_l & \text{if } l.r[A] \in l.q[B] \\ \llbracket C \rrbracket_l & \text{otherwise} \end{cases} \end{aligned}$$

DEFINITION 11 (Endpoint Projection). Let  $C$  be a term of AC. The endpoint projection of  $C$ , denoted by  $\llbracket C \rrbracket$ , is defined as:

$$\llbracket C \rrbracket = \prod_{p \in \text{fp}(C)} \llbracket C \rrbracket_p \mid \prod_l \left( \bigsqcup_{p \in \llbracket C \rrbracket_l} \llbracket C \rrbracket_p \right)$$

The EPP of a choreography  $C$  is the composition of the projections of all active processes and the merged projections of the service processes in  $C$ .

### 4.1 Projection Example

As an example, we project the choreography  $C = C_c \mid C_s$  presented in § 2.3. Applying the definition of EPP to  $C$  we obtain

$$\llbracket C \rrbracket = \llbracket C \rrbracket_c \mid \llbracket C \rrbracket_f \mid \llbracket C \rrbracket_a \mid \llbracket C \rrbracket_{dm} \mid \llbracket C \rrbracket_l$$

Figure 9 reports some excerpts of  $\llbracket C \rrbracket$ , i.e.,  $\llbracket C \rrbracket_a$ ,  $\llbracket C \rrbracket_{dm}$ , and  $\llbracket C \rrbracket_l$ . We illustrate how  $\llbracket C \rrbracket$  implements the same behaviour of  $C$ .

**Start.** The start of a new session, like the one at Lines 1 of  $C_c$  and  $C_s$ , applies the same effect in the projection. Let  $\llbracket C \rrbracket'_p$  be the continuation of the process projection of  $C$  on  $p$  after Line 1. The execution of Lines 1 of the process projections composing  $\llbracket C \rrbracket$  is

$$\begin{aligned} &\text{req } k_d : c[C] \Leftrightarrow l_A.a[A], l_{DM}.dm[DM], l_L.l[L]; \llbracket C \rrbracket'_c \mid \\ D, &\text{acc } k_d : l_A.a[A]; \llbracket C \rrbracket'_a \mid \text{acc } k_d : l_{DM}.dm[DM]; \llbracket C \rrbracket'_s \mid \rightarrow \\ &\text{acc } k_d : l_L.l[L]; \llbracket C \rrbracket'_l \mid \llbracket C \rrbracket'_f \\ D', &\llbracket C \rrbracket'_c \mid \llbracket C \rrbracket'_a \mid \llbracket C \rrbracket'_{dm} \mid \llbracket C \rrbracket'_l \mid \llbracket C \rrbracket'_f \end{aligned}$$

Let the location of process  $c$  be  $l_c$  in  $D$ . The effect of the start is  $\delta = \text{start } k_d : l_c.c[C], l_A.a[A], l_{DM}.dm[DM], l_L.l[L]$  which can generate the same  $D'$  as the start at Lines 1 of  $C_c$  and  $C_s$ .

**Communications.** The EPP projects a complete communication to a partial send for the sender and a partial receive for the receiver. Line 6 of  $C_s$  is  $k_d : dm[DM].\text{logok}(\text{rsc}) \rightarrow l_L.\text{log}(\text{log})$  and in  $\llbracket C \rrbracket$  it is projected into a partial send for  $dm$  (at Line 4 of  $\llbracket C \rrbracket_{dm}$ ) and a partial receive for  $l$  (at Line 2 of  $\llbracket C \rrbracket_l$ ). The semantics of Line 6 of  $C_s$  and that of Line 4 of  $\llbracket C \rrbracket_{dm}$  and Line 2 of  $\llbracket C \rrbracket_l$  is equal as the complete action breaks into a partial send, equal to the effect of Line 4 of  $\llbracket C \rrbracket_{dm}$ , and continues as a partial receive, equal to  $\llbracket C \rrbracket_l$ .

**Conditionals.** The EPP of a conditional merges the behaviour of all processes in its branches as branched receptions. For example, at Line 3 of  $C_s$ , process  $a$  evaluates a condition that either branches in the behaviour described at Lines 4-15 or at Lines 17-19. At Line 3 of  $\llbracket C \rrbracket_a$  the conditional is preserved verbatim. For the other processes present in the branches of the conditional, the EPP merges their behaviours on a branched reception:  $\llbracket C \rrbracket_{dm}$  projects the conditional on the branches guarded by operations *ok* or *ko* (resp. at Lines 3 and 6).  $\llbracket C \rrbracket_l$  merges the branches of the conditional into a single branch on operation *log*, at Line 2.

### 4.2 Properties

We present the properties of EPP. Basically, we prove that the EPP implements the same behaviour of its originating choreography. To do that, we build on the foundation that the EPP of a choreography is still typable and then we establish a bisimilarity relation between the semantics of the EPP and the originating choreography.

First we state our type preservation result using the *minimal typing* of choreographies  $\vdash_{\min}$ , which types the branches in rules

$$\begin{aligned}
\llbracket C \rrbracket_a &= \begin{cases} 1. \text{acc } k_d : l_a.a[A]; \\ 2. k_d : C \rightarrow a[A].get(\underline{\text{req}}); \\ 3. \text{if } a.isValid(\underline{\text{req}}) \{ \\ 4. \quad k_d : a[A].\underline{\text{req.rsc}} \rightarrow S.ok; \\ 5. \quad k_d : a[A] \rightarrow C.ok \\ 6. \} \text{ else } \{ \\ 7. \quad k_d : a[A].\underline{\text{req.rsc}} \rightarrow S.ko; \\ 8. \quad k_d : a[A] \rightarrow C.ko \\ 9. \} \end{cases} & \llbracket C \rrbracket_{dm} = \begin{cases} 1. \text{acc } k_d : l_s.dm[DM]; \\ 2. k_d : A \rightarrow dm[DM].\{ \\ 3. \quad ok(\underline{\text{rsc}}); \\ 4. \quad k_d : dm[DM].\logok(\underline{\text{req}}) \rightarrow L.log; \\ 5. \quad \dots \\ 6. \quad ko(\underline{\text{rsc}}); \\ 7. \quad k_d : dm[DM].\logko(\underline{\text{req}}) \rightarrow L.log \\ 8. \} \end{cases} & \llbracket C \rrbracket_l = \begin{cases} 1. \text{acc } k_d : l_l.l[L]; \\ 2. k_d : S \rightarrow l[L].\log(\underline{\text{log}}); \end{cases}
\end{aligned}$$

Figure 9: Endpoint Projections of  $C_c \mid C_s$ , from Figure 4. Process projection on processes a, dm (excepts), and l.

$$\begin{aligned}
S &::= \langle B_s, P \rangle_l \quad (\text{service}) \quad | \quad S \mid S' \quad (\text{network}) \\
B_s &::= ! (x); B \quad (\text{accept}) \quad | \quad \mathbf{0} \quad (\text{inact}) \\
P &::= B \cdot t \cdot M \quad (\text{process}) \quad | \quad P \mid P' \quad (\text{par}) \\
B &::= \sum_i [o_i(x_i) \text{ from } e] \{B_i\} \quad (\text{choice}) \\
&\quad \left| \begin{array}{ll} o(x) \text{ from } e; B & (\text{input}) \\ o@e_1(e_2) \text{ to } e_3; B & (\text{output}) \\ \text{if } e \{B_1\} \text{ else } \{B_2\} & (\text{cond}) \\ \text{def } X = B' \text{ in } B & (\text{def}) \end{array} \right| \begin{array}{ll} ?@e_1(e_2); B & (\text{request}) \\ \text{cq}(x); B & (\text{cqueue}) \\ x = e; B & (\text{assign}) \\ X & (\text{call}) \\ \mathbf{0} & (\text{inact}) \end{array}
\end{aligned}$$

Figure 10: Correlation Calculus, syntax

$[T]_{\text{SEND}}$  and  $[T]_{\text{RECV}}$  using the respective minimal branch types. As in [8],  $\vdash_{\min}$  takes into account that, due to merging, the EPP of a complete choreography may still offer branches that the originating choreography discarded with a conditional. Thus we have:

**THEOREM 4 (EPP Typing Preservation).**

$$\Gamma \vdash_{\min} D, C \text{ implies } \llbracket \Gamma \rrbracket \vdash_{\min} D, \llbracket C \rrbracket$$

Where  $\llbracket \Gamma \rrbracket$  replaces the typings of recursive procedures in  $\Gamma$  with the typing of each procedure at each endpoint process taking part in it.

Combining the above Theorem with Theorem 2 we can prove:

**THEOREM 5 (EPP Operational Correspondence).**

Let  $D, C$  be well-typed. Then,

1. (Completeness)  $D, C \rightarrow D', C'$  implies  $D, \llbracket C \rrbracket \rightarrow D', C''$  and  $\llbracket C' \rrbracket \prec C''$ .
2. (Soundness)  $D, \llbracket C \rrbracket \rightarrow D', C'$  implies  $D, C \rightarrow D', C''$  and  $\llbracket C'' \rrbracket \prec C'$ .

In the theorem above  $C \prec C'$  is the *pruning relation*, a strong typed bisimilarity [9] such that  $C$  has some unused branches and always-available accepts.

## 5. Dynamic Correlation Calculus Language

We introduce the Dynamic Correlation Calculus (DCC), the target language of our compilation, modelled on the Correlation Calculus [26] (CC) and extended to support the dynamic creation of queues. The reason to target CC is that it is the formal model of the executable language Jolie [21]. In this way, our results are immediately applicable to an actual implemented language, yet preserving the formality and simplicity of a theoretical approach. Moreover, the semantics of CC formalises message correlation à-la SOC, as in the standard service-oriented language BPEL [30], by following similar concepts to those that we used in our semantics for AC. However, it is complex to implement multiparty sessions in CC because processes are statically linked to a single queue and, through it, to a single correlation key. For these reasons, we extend CC to DCC by allowing processes to create queues at runtime and selectively read messages from them.

**Syntax.** The syntax of DCC (in Figure 10) is divided in two layers: *services*, ranged over by  $S$ , and *processes*, ranged over by  $P$ .

The term (*service*) describes a service, located at  $l$ , as a container of a *start behaviour*  $B_s$  and a (system of) processes  $P$ . A start behaviour allows to create new processes on request: following the syntax of  $B_s$ , a service can spawn a new process that stores the message of the request under the bound variable  $x$  and implements the behaviour  $B$  (see below). The term (*network*) supports interactions among services. The process layer  $P$  defines the structure of processes, which run inside of services, and their composition. In the syntax, a process is the association of a behaviour  $B$ , a state  $t$ , and a queue map  $M$ . The state  $t$  and the map  $M$  are defined exactly like the states and queue maps found in AC (§ 2). Still from AC, we also make use of names for operations ( $o$ ), procedures ( $X$ ), and variables ( $x$ , which we recall are paths). Expressions, ranged over by  $e$ , are evaluated at runtime on the state of the related process. DCC models communications with terms (*input*) and (*output*). In (*input*), the process stores in  $x$  a message from the head of the queue correlating with  $e$  and expectedly received on operation  $o$ . Dually, term (*output*) describes the delivery of a message on operation  $o$  with content  $e_2$ . In the term,  $e_1$  defines the location of the service where the addressee (process) is running, whilst  $e_3$  is the key that correlates with the receiving queue of the addressee. The term (*choice*) models a branching input-choice. The argument of the choice is modelled after (*input*). If one of the inputs can execute — receiving a message from the queue correlating with  $e$  on operation  $o_i$  — it discards all other receptions and executes the related behaviour  $B_i$ . Term (*request*) is the dual of (*accept*) and asks the service located at  $e_1$  to spawn a new process, passing to it the message in  $e_2$ . Finally (*cqueue*) models the creation of a new queue. After the creation, variable  $x$  contains the key that correlates with the new queue. All other terms of the syntax are standard.

**Semantics.** In Figure 11 we report an excerpt of the semantics of DCC, given as rules for a reduction relation  $\rightarrow$ . Relation  $\rightarrow$  is closed under a structural congruence  $\equiv$  defined in the standard way; in particular, it supports commutativity and associativity for parallel composition. We comment the rules. Rule  $[\text{DCC}]_{\text{RECV}}$  models the reception of messages: if the queue correlating with  $e$  has a message on operation  $o$  then the message is removed from the queue and its content is assigned to the variable  $x$  in the state of the process. Rule  $[\text{DCC}]_{\text{CQ}}$  adds to  $M$  an empty queue ( $\varepsilon$ ) correlating with a fresh key, stored in  $x$ . The key is unique within the service of the process, to avoid ambiguity, but we impose no requirements on its structure (it can be any tree as long as it is unique in the enclosing service). Rule  $[\text{DCC}]_{\text{SEND}}$  models the delivery of a message between processes in different services. In the rule, the message from the sender is added to (the end of) the correlating queue of the receiver. Similarly, rule  $[\text{DCC}]_{\text{START}}$  models the matching of a request to create a new process with the service that accepts it. The newly created process has the defined for the service, a state initialised with the content of the request message, and an empty queue map.

## 6. Compiler from AC to DCC and Properties

We formalise our main contribution: the definition of a formally-correct compiler from AC to DCC, which models how applied choreographies can be implemented in real-world languages.

$$\begin{array}{c}
\frac{t_c = \text{eval}(e, t) \quad M(t_c) = (o, t') :: \tilde{m}}{o(x) \text{ from } e; B \cdot t \cdot M \rightarrow B \cdot t \triangleleft (x, t') \cdot M[t_c \mapsto \tilde{m}]} \text{[DCC]}_{\text{RECV}} \quad \frac{P = \text{cq}(x); B \cdot t \cdot M \quad t_c \notin \bigcup_i \text{dom}(M_i) \cup \text{dom}(M) \quad t' = t \triangleleft (x, t_c)}{\langle B_s, P \mid \prod_i B_i \cdot t_i \cdot M_i \rangle_l \rightarrow \langle B_s, B \cdot t' \cdot M[t_c \mapsto \varepsilon] \mid \prod_i B_i \cdot t_i \cdot M_i \rangle_l} \text{[DCC]}_{\text{CQ}} \\
\\
\frac{P = o@e_1(e_2) \text{ to } e_3; B \cdot t \cdot M \quad \text{eval}(e_1, t) = l' \quad \text{eval}(e_2, t) = t_m \quad \text{eval}(e_3, t) = t_c \quad M'' = M'[t_c \mapsto M'(t_c) :: (o, t_m)]}{\langle B_s, P \mid P_1 \rangle_l \mid \langle B'_s, B' \cdot t' \cdot M' \mid P_2 \rangle_{l'} \rightarrow \langle B_s, B \cdot t \cdot M \mid P_1 \rangle_l \mid \langle B'_s, B' \cdot t' \cdot M'' \mid P_2 \rangle_{l'}} \text{[DCC]}_{\text{SEND}} \\
\\
\frac{P_1 = ?@e_1(e_2); B_1 \cdot t_1 \cdot M_1 \quad \text{eval}(e_1, t_1) = l \quad \text{eval}(e_2, t_1) = t_m \quad Q = B \cdot t_\perp \triangleleft (x, t_m) \cdot \emptyset}{\langle ! (x); B, P \rangle_l \mid \langle B'_s, P_1 \mid P_2 \rangle_{l'} \rightarrow \langle ! (x); B, Q \mid P_l \rangle_l \mid \langle B'_s, B_1 \cdot t_1 \cdot M_1 \mid P_2 \rangle_{l'}} \text{[DCC]}_{\text{START}}
\end{array}$$

Figure 11: Correlation Calculus, semantics — selected rules

## 6.1 Compiler

$\boxed{D, C}^\Gamma$  denotes the compilation of a well-typed running choreography  $D, C$  into an operationally-equivalent network of DCC services. Before giving the formal definition of  $\boxed{D, C}^\Gamma$ , we need to introduce some additional notation. Let  $\mathcal{C}[\cdot]$  be an AC context defined as usual:  $\mathcal{C}[\cdot] ::= \cdot \mid \mathcal{C}[\cdot] \mid C \mid C \mid \mathcal{C}[\cdot]$ . We then formalise the filtering operators  $C|_l$  and  $C|_p$  as follows:

$$\begin{aligned}
C|_l &= C' \text{ if } C = \mathcal{C}[C'] \text{ and } C' = \mathbf{acc} \ k : l.p[A]; C'', \mathbf{0} \text{ otherwise} \\
C|_p &= C' \text{ if } C = \mathcal{C}[C'] \text{ and } \{p\} = \text{fp}(C'), \mathbf{0} \text{ otherwise}
\end{aligned}$$

Intuitively,  $C|_l$  returns the accept term at location  $l$  in  $C$  and  $C|_p$  returns the endpoint choreography of process  $p$  in  $C$ . Next, we denote with  $\boxed{C}^\Gamma$  the compilation of a process projection defined on the rules reported in Figure 12 (and commented below). Below, we define our compiler abusing the notation  $l \in \Gamma$  to say that  $p@l \in \Gamma$  for some  $p$  or that there is a service typing in  $\Gamma$  containing  $l$ .

**DEFINITION 12 (Compilation).** Let  $C$  be a parallel composition of endpoint choreographies and assume that  $\Gamma \vdash D, C$  for some  $\Gamma$  and  $D$ . Then, the compilation  $\boxed{D, C}^\Gamma$  is defined as:

$$\boxed{D, C}^\Gamma = \prod_{l \in \Gamma} \left\langle \boxed{C|_l}^\Gamma, \prod_{p \in D(l)} \boxed{C|_p}^\Gamma \cdot D(p).\text{st} \cdot D(p).\text{que} \right\rangle_l$$

Intuitively, for each service  $\langle B_s, P \rangle_l$  in the compiled network: *i*) the start behaviour  $B_s$  is the compilation of the endpoint choreography in  $C$  accepting the creation of processes at location  $l$ ; *ii*)  $P$  is the parallel composition of the compilation of all active processes located at  $l$ , equipped with their respective states and queue maps according to  $D$ . Let us now comment the rules in Figure 12. We use the auxiliary notation  $\odot$  for DCC behaviours, defined as

$$\odot (B_i) = B_1; \dots; B_n.$$

$i \in [1, n]$

**Requests.** Function *start* defines the compilation of (*req*) terms. *start* compiles (*req*) terms to create the queues and a part of the session descriptor of a valid session support (see Definition 2) for the starter. Given a session identifier  $k$ , the located role of the starter ( $l_A.A$ ), and the other located roles in the session ( $l_B.B$ ), *start* returns DCC code that: ( $s_1$ ) includes in the Session Descriptor all the locations of the processes involved in the session and ( $s_{2.1}$ ) all the keys correlating with the queues of the starter for the session. Then, ( $s_{2.2}$ ) it requests the creation of all the service processes for the session, ( $s_{2.3}$ ) waits for them to be ready using the reserved operation *sync* and, finally, ( $s_3$ ) sends them the complete session descriptor obtained after receiving from all processes their correlation keys (in the *sync* step).

**Accepts.** (*acc*) terms define the start behaviour of the service accepting the creation of processes at a location. Given a session descriptor  $k$ , the role  $B$  of the service process, and the service typing  $G \langle A | \tilde{C} | \tilde{D} \rangle$  of the location, function *accept* defines the compilation of (*acc*) terms. *accept* complements function *start* by compiling the code that ( $a_1$ ) accepts the spawn of a new process which, in turn, ( $a_2$ ) creates its queues (including their keys in the Session

Descriptor passed by the starter), ( $a_3$ ) returns the Session Descriptor to the starter, and ( $a_4$ ) waits for the signal to start the session.

**Other terms.** We compile (*send*) terms to (*output*) terms. Observe that in the syntax of the compiled code there are the same elements used by the semantics of AC to implement correlation, i.e., the location of the receiver ( $k.B.l$ ) and the key that correlates with its queue ( $k.A.B$ ). (*recv*) compiles to (*choice*), which defines the path ( $k.A.B$ ) of the key correlating with the receiving queue, used also in the semantics of AC. Terms (*def*), (*cond*), (*call*), and (*inact*) compile to the relative terms in DCC.

We report in Figure 13 the compilation of the example in § 2.3.

## 6.2 Properties

We present the main results of this paper: the operational correspondence between the semantics of a well-typed running Applied Choreography and the semantics of its compilation.

**THEOREM 6 (Compilation Operational Correspondence).**

Let  $C$  be a composition of endpoint choreographies such that  $\Gamma \vdash D, C$ . Then we have that:

1. (Completeness)  $D, C \rightarrow D', C'$  implies  $\boxed{D, C}^\Gamma \rightarrow^+ \boxed{D', C'}^{\Gamma'}$  for some  $\Gamma'$  such that  $\Gamma' \vdash D', C'$ .
2. (Soundness)  $\boxed{D, C}^\Gamma \rightarrow^* S$  implies
  - (i)  $D, C \rightarrow^* D', C'$  and (ii)  $S \rightarrow^* \boxed{D', C'}^{\Gamma'}$  for some  $D', C'$  and  $\Gamma'$  such that  $\Gamma' \vdash D', C'$ .

Theorem ?? assumes that we are dealing with endpoint choreographies because the compiler is defined only for such terms. Using EPP, we can always reconduce a choreography to this case.

**LEMMA 1** Let  $C$  be well-typed. Then,  $\llbracket C \rrbracket$  is a composition of endpoint choreographies.

We can now conclude by answering our research question in the Introduction. The compilation of the EPP of an applied choreography is always a correct implementation based on message correlation.

**COROLLARY 1 (Applied Choreographies).**

Let  $\Gamma \vdash D, C$ . Then we have that:

1. (Completeness)  $D, C \rightarrow D', C'$  implies  $\llbracket D, C \rrbracket^\Gamma \rightarrow^+ \llbracket D', C' \rrbracket^{\Gamma'}$  for some  $\Gamma'$  such that  $\Gamma' \vdash D', C'$ .
2. (Soundness)  $\llbracket D, C \rrbracket^\Gamma \rightarrow^* S$  implies
  - (i)  $D, C \rightarrow^* D', C'$  and (ii)  $S \rightarrow^* \llbracket D', C' \rrbracket^{\Gamma'}$  for some  $D', C'$  and  $\Gamma'$  such that  $\Gamma' \vdash D', C'$ .

## 7. Related Work and Discussion

**Implementation Model.** To the best of our knowledge, this is the first work on a theory for a compiler and an execution model for choreographies. All previous works on formal choreography languages only specify an EPP procedure towards a calculus based on name synchronisation leaving the design of its concrete support to implementors.

$$\begin{array}{ll}
\boxed{\text{req}} k : p[A] \Leftrightarrow \widetilde{l.B}; C \Gamma & = \text{start}(k, l'.A, \widetilde{l.B}); \boxed{C} \Gamma, p@l' \in \Gamma & \boxed{\text{acc}} k : l.q[B]; C \Gamma & = \text{accept}(k, B, \Gamma(\widetilde{l})); \boxed{C} \Gamma, l \in \widetilde{l}, \widetilde{l} \in \Gamma \\
\boxed{k : p[A].e \rightarrow B.o; C} \Gamma & = o@k.B.l(e) \text{ to } k.A.B; \boxed{C} \Gamma & \boxed{k : A \rightarrow q[B].\{o_i(x_i); C_i\}_{i \in I}} \Gamma & = \sum_{i \in I} [o_i(x_i) \text{ from } k.A.B] \{ \boxed{C_i} \Gamma \} \\
\boxed{\text{def } X = C' \text{ in } C} \Gamma & = \text{def } X = \boxed{C'} \Gamma \text{ in } \boxed{C} \Gamma & \boxed{\text{if } p.e \{C_1\} \text{ else } \{C_2\}} \Gamma & = \text{if } e \{ \boxed{C_1} \Gamma \} \text{ else } \{ \boxed{C_2} \Gamma \} \\
\boxed{X} \Gamma & = X & \boxed{0} \Gamma & = 0
\end{array}$$

$$\begin{array}{l}
\text{start}(k, l_A.A, \widetilde{l_B.B}) = \overbrace{\begin{array}{c} \textcircled{\bullet} \text{ k.I.l} = l_I \\ \text{I} \in \{A, \widetilde{B}\} \end{array}}^{(s_1) \text{ locations}} \overbrace{\begin{array}{c} \textcircled{\bullet} \left( \text{cq}(\text{k.I.A}) ; \text{?@k.I.l(k)} ; \text{sync(k) from k.I.A} \right) \\ \text{I} \in \{\widetilde{B}\} \end{array}}^{(s_2) \text{ creation of correlation keys and service processes}} \overbrace{\textcircled{\bullet} \text{ start@k.I.l(k) to k.A.I}}^{(s_3) \text{ session start}} \\
\text{accept}(k, B, G(A|\widetilde{C}|\widetilde{D})) = \overbrace{\textcircled{\bullet} \text{ k.I.l} = l_I}^{(a_1) \text{ spawn accept}} ; \overbrace{\textcircled{\bullet} \left( \text{cq}(\text{k.I.B}) \right)}^{(a_2) \text{ create correlation keys and queues}} ; \overbrace{\text{sync@k.A.l(k) to k.B.A}}^{(a_3) \text{ service response}} ; \overbrace{\text{start(k) from k.A.B}}^{(a_4) \text{ session start}} \\
\text{I} \in \{A, \widetilde{C}\} / \{\widetilde{B}\}
\end{array}$$

Figure 12: Compiler from AC to DCC.

$$\begin{array}{l}
\overbrace{\langle \boxed{C}_f \Gamma, \boxed{C}_c \Gamma, D(c).st \cdot D(c).que \rangle_{l_c}}^{S_c} \mid \overbrace{\langle \boxed{C}_a \Gamma, 0 \rangle_{l_a}}^{S_a} \mid \overbrace{\langle \boxed{C}_{dm} \Gamma, 0 \rangle_{l_{dm}}}^{S_{dm}} \mid \overbrace{\langle \boxed{C}_l \Gamma, 0 \rangle_{l_l}}^{S_l} \\
\boxed{C}_c \Gamma = \begin{cases} 1. \text{kd.C.l} = l_c; \\ 2. \text{kd.A.l} = l_a; \\ 3. \text{kd.DM.l} = l_{dm}; \\ 4. \text{kd.L.l} = l_l; \\ 5. \text{cq}(\text{kd.A.C}); \\ 6. \text{?@kd.A.l(kd)}; \\ 7. \text{sync(kd) from kd.A.C}; \\ 8. \dots \\ 9. \dots \\ 10. \text{start@kd.A.l(kd) to kd.C.A}; \\ 11. \text{start@kd.S.l(kd) to kd.C.S}; \\ 12. \text{start@kd.C.l(kd) to kd.C.L}; \\ 13. \text{get@kd.A.l(mkReq()) to kd.C.A}; \\ 14. \dots \end{cases} \quad \boxed{C}_a \Gamma = \begin{cases} 1. \text{!(kd)}; \\ 2. \text{cq}(\text{kd.C.A}); \\ 3. \text{cq}(\text{kd.S.A}); \\ 4. \text{cq}(\text{kd.L.A}); \\ 5. \text{sync@kd.C.l(kd) to kd.A.C}; \\ 6. \text{start(kd) from kd.C.A}; \\ 7. \text{get(kd) from kd.C.A}; \\ 8. \text{if isValid(req) \{ } \\ 9. \quad \text{ok@kd.S.l(req.rsc) to kd.A.S}; \\ 10. \quad \text{ok@kd.C.l() to kd.A.C} \\ 11. \} \text{ else \{ } \\ 12. \quad \text{ko@kd.S.l(req.rsc) to kd.A.S}; \\ 13. \quad \text{ko@kd.C.l() to kd.A.C} \\ 14. \} \end{cases} \quad \boxed{C}_l \Gamma = \begin{cases} 1. \text{!(kd)}; \\ 2. \text{cq}(\text{kd.C.L}); \\ 3. \text{cq}(\text{kd.A.L}); \\ 4. \text{cq}(\text{kd.S.L}); \\ 5. \text{sync@kd.C.l(kd) to kd.L.C}; \\ 6. \text{start(kd) from kd.C.L}; \\ 7. \text{log(log) from kd.S.L} \end{cases}
\end{array}$$

Figure 13: Compilation of  $C = C_c \mid C_s$ , from Figure 4. Compiled behaviour of processes a, c (excepts) and l.

Chor and AIOCI [2, 12] are the only projects (that we are aware of) that aim at providing choreography languages with strong safety guarantees (e.g., deadlock-freedom). The languages respectively implement the formal models presented in [8] and [33]. However, as already mentioned, both implementations depart significantly from their respective formal models as they generate code based on message correlation whilst their formalisations of EPP use synchronisations on names. This gap between theoretical models and their implementations has two consequences: *i*) it breaks the correctness-by-construction guarantee of choreographies, providing no proof that the implementation correctly supports synchronisations on names and *ii*) users must look into the complexity of the compilers to understand how the generated code implements the originating choreography. Implementations of other frameworks based on sessions share similar issues and follow different custom practices to implement the semantics of name synchronisation [19, 20, 29]. Our work is thus a useful reference to formalise the implementation of session-based languages in general.

**Choreography Language and Deployment.** The syntax of AC resembles that of Compositional Choreographies [27], which introduced a notion of compositionality in choreographies. This similarity is intentional: our aim is to show that it is possible to provide a suitable implementation model for this kind of languages. Also, the fragment of endpoint choreographies in AC is remarkably similar to standard process languages based on sessions, e.g., those used in [9, 15–17]. The key point of departure wrt these works is that we use deployments to support communications with message correlation rather than name synchronisation.

An interesting aspect of our semantics is that it captures asynchronous message passing through the interplay of the swapping relation  $\simeq_c$ , partial choreographies, and message queues. By contrast, previous choreography models dealing with asynchrony come with ad-hoc rules that check whether actions in a continuation guarded by a communication (an  $\eta$ ) should be allowed in order to simulate asynchrony [8, 27]. This requires labelling reductions with the participants of actions, which is not necessary in our setting. In general, our choreography calculus is more modular than previous proposals: by changing the definitions of deployments and effects ( $D, \delta \triangleright D'$ ), we could formalise different communication semantics (e.g., synchronous or asynchronous with buffers) without changing the other rules.

**Delegation.** Delegation is a session mobility mechanism for transferring the responsibility to continue a session from a process to another, a standard feature of session-typed choreography models [8, 27]. It would be easy to add delegation to Applied Choreographies, by adding a rule that atomically updates the session descriptors of all processes involved in a session when there is a delegation. We chose to leave delegation to future work, since its introduction would introduce more complexity in our compiler; specifically, in DCC, we would have to compile appropriate communications for updating the local session descriptors of participants. This is a widely-known difficulty of implementing delegation; we plan on addressing this limitation by formalising the techniques proposed in the implementations given in [8, 19].

**Correlation keys.** In the semantics of AC, we abstracted from how unique (wrt locations, i.e., services) correlation keys are generated. This loose definition is intentional: it allows our model to capture

a broad spectrum of implementations, provided that they satisfy the requirement of unicity of keys. As future work, we plan to implement a language based on AC that allows the definition of custom procedures for the generation of correlation keys (e.g., from database queries, cookies, UUIDs, etc.).

## Acknowledgements

Montesi was supported by CRC (Choreographies for Reliable and efficient Communication software), grant no. DFF-4005-00304 from the Danish Council for Independent Research.

## References

- [1] Business Process Model and Notation. <http://www.omg.org/spec/BPMN/2.0/>.
- [2] AIOCI. Programming Language. <http://www.cs.unibo.it/projects/jolie/aioej.html>.
- [3] S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In *POPL*, pages 191–202, 2012.
- [4] L. Bettini, M. Coppo, L. D’Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
- [5] T. Bray. The javascript object notation (json) data interchange format. 2014.
- [6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210*. <http://www.w3.org/TR/1998/REC-xml-19980210>, 16, 1998.
- [7] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of the 7th International Conference on Coordination Models and Languages (COORDINATION’06)*, volume 4038 of *LNCS*, pages 63–81, Heidelberg, Germany, 2006. Springer-Verlag.
- [8] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
- [9] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
- [10] M. Carbone, F. Montesi, and C. Schürmann. Choreographies, logically. In *CONCUR*, pages 47–62, 2014.
- [11] S. Carpineti, C. Laneve, and P. Milazzo. Bopi - A distributed machine for experimenting web services technologies. In *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005)*, 6-9 June 2005, St. Malo, France, pages 202–211, 2005. . URL <http://dx.doi.org/10.1109/ACSD.2005.6>.
- [12] Chor. Programming Language. <http://www.chor-lang.org/>.
- [13] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 760:1–65, 2015.
- [14] P. Deniérou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, pages 174–186. Springer, 2013. . URL [http://dx.doi.org/10.1007/978-3-642-39212-2\\_18](http://dx.doi.org/10.1007/978-3-642-39212-2_18).
- [15] S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, Nov. 2005.
- [16] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP’98*, volume 1381 of *LNCS*, pages 22–138, Heidelberg, Germany, 1998. Springer-Verlag.
- [17] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, volume 43(1), pages 273–284. ACM, 2008.
- [18] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *Proc. of ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.
- [19] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *ECOOP*, pages 516–541, 2008.
- [20] R. Hu, R. Neykova, N. Yoshida, R. Demangeon, and K. Honda. Practical interruptible conversations - distributed dynamic verification with session types and python. In *RV*, pages 130–148, 2013.
- [21] Jolie. Programming Language. <http://www.jolie-lang.org/>.
- [22] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Proc. of SEFM*, pages 323–332. IEEE, 2008.
- [23] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
- [24] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, Sept. 1992.
- [25] F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. <http://www.itu.dk/people/fabr/papers/phd/thesis.pdf>.
- [26] F. Montesi and M. Carbone. Programming services with correlation sets. In *ICSOC*, pages 125–141, 2011.
- [27] F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, pages 425–439, 2013.
- [28] F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. 2014.
- [29] R. Neykova and N. Yoshida. Multiparty session actors. In *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 131–146, 2014.
- [30] B. OASIS. Web services business process execution language. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>, 2007.
- [31] PI4SOA. <http://www.pi4soa.org>, 2008.
- [32] B. C. Pierce. *Types and Programming Languages*. MIT Press, MA, USA, 2002.
- [33] M. D. Preda, M. Gabbriellini, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies - safe runtime updates of distributed applications. In *COORDINATION*, pages 67–82, 2015.
- [34] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982, United States, 2007. IEEE Computer Society Press.
- [35] Savara. JBoss Community. <http://www.jboss.org/savara/>.
- [36] W3C WS-CDL Working Group. Web services choreography description language version 1.0. <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>, 2004.